Project no. FP6-028038

Palette

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

Integrated Project

Technology-enhanced learning

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

D.INF.01: Report on the design of extension mechanisms for creating templates, using templates for editing and customizing the user interface, and of extensions to be integrated in the information reuse tool (M3)

Due date of deliverable: April 30, 2006
Actual submission date: May 15, 2006

Organisation name of lead contractor for this deliverable: INRIA

Final version

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
R	Public	**PU**

Keywords: Information services, structured documents, document reuse, document editing, template languages, document transformations, automatic transformations, multimedia, XML, SMIL, XHTML, XML Schema, schema matching, structure matching.

Authors: A. Boukottaya (EPFL), F. Campoy Flores (INRIA), R. Deltour (INRIA), V. Quint (INRIA), Ch. Vanoirbeek (EPFL), I. Vatton (INRIA), K. Zouba (EPFL)

Summary

In order to support the activities of participants in Communities of Practice, the Palette project will provide tools for document production and for document reuse in heterogeneous applications. The aim is to reduce the current limitations caused by the proliferation of data sources deploying a variety of modalities, information models and encoding syntaxes. This will enhance applicability and performances of document technologies within pedagogically consistent scenarios.

To achieve this goal, two directions are taken: 1) the adoption of well established standards and 2) the development of tools specifically suited for use in CoPs.

The chosen standards for document formats are based on XML technologies. Documents using discrete media such as text, pictures, graphics and mathematical expressions are represented in XHTML, SVG and MathML, under the form of compound documents, while multimedia documents involving continuous media such as video, sound and animations use the SMIL language. A number of tools handle these formats, thus offering a wide range of applications to CoP participants. Two kinds of tools are developed for two different tasks: document authoring and document reuse.

The authoring tools are based on the principle of separating the authoring model from the publication format. Most document editors exhibit a user interface that is mainly determined by the features offered by the format. The approach developed here is different. It aims at allowing users to create the authoring model that best fills the requirements of the specific documents they have to produce, while generating documents in a standard format. This approach is applied to both discrete and continuous media, through two different models introduced below.

For compound documents using discrete media, a generic structure description language was designed. It allows the particular structure of a type of document to be described in terms of the standard document format language(s). This language, called XTiger, is both simple and powerful. It defines only the special parts of the document structure that makes the document type unique, and it relies on the underlying languages (XHTML, SVG, MathML) for the less specific parts.

For multimedia documents published in SMIL, two steps are necessary. First an object model is used to represent generic objects with all their facets, i.e. the logical, temporal and spatial dimensions. Then, a template model describes how these objects have to be combined and specialized to make a particular type of multimedia document.

The document reuse tool is based on structure transformations. To allow a document produced in a given XML language to be handled by an application accepting a different XML language, a structure transformation has to be performed. This is usually done by describing the transformation in the XSLT language. The reuse tool aims at assisting the user in generating this transformation. It first compares the schemas of the source and result structures and, based on the matching of one structure onto the other, it generates the XSLT transformation.

The implementation of these techniques will be done in three different tools, each based on software that is already available. The XTiger language will constitute an extension of the Amaya editor. The multimedia authoring model will be implemented in the next version of the LimSee editor. The document reuse tool will be based on a previous prototype.

Contents

1. Introduction

WP2 of Palette is dedicated to Information Services. It aims at providing services for information production and reuse. It focuses on structured multimedia documents and on tools for producing and using this type of document. Authoring tools and information reuse tools are in the focus of this WP.

For authoring tools, the objective is to provide users with a flexible, adaptable environment that allows them to efficiently produce different types of multimedia documents, specifically targeted for use in communities of practice. The authoring tools are based on existing software platforms already available at INRIA, Amaya and LimSee2.

For structured documents reuse, the objective is to allow existing documents initially structured for a given purpose to be repurposed for use in a different context. This will improve (re)usability of information in communities of practice. The document reuse tool is based on a research prototype recently developed at EPFL.

The first task of WP2 consists in preparing the development of these tools by defining the software architecture, more specifically the extension of the architecture of the existing tools that will enable the development of the new services. For the authoring tools, the main issue is to support a mechanism that enables templates and high-level document models. For the information reuse tool, two major issues have to be addressed: enhancing robustness and improving the user interface.

All the tools considered here are based on the structured document approach. With this approach, a document is represented as a structure of abstract elements. The structure includes hierarchical relationships and hypertext links. The elements assembled by the structure are typically titles, paragraphs, sections, subsections, headings, figures, lists, references, tables, table rows, table cells, etc. Different types of documents involve different elements, assembled in different ways. For this reason, each document type can be specified by a generic structure (also called schema or document type definition – DTD) which defines the available elements and the rules for making the structure. Document instances follow the generic structure of their type.

This approach brings a number of advantages. Documents represented that way may be used in many different types of applications. Multiple graphical representations may be produced by applying different styles to the structure. Other documents may be automatically derived from an original document by selecting some parts of the original structure and re-arranging them in a different way. Information retrieval is facilitated by the explicit structure. By conveying a rich representation of information, rigorously defined by a schema or a DTD, structured documents open a broad scope of possible treatments.

The concepts of structured documents have been standardized in XML [Bray 04a] and a series of associated technologies that are now widely accepted. These standards were developed initially for the web by W3C, but their range of application is now much broader. The tools presented here are based on these standards.

This report presents the achievements of the architecture task of WP1. As information services in Palette are based on three different software platforms, the rest of this report is organized in three part, one for each platform.

2. Templates in Amaya

2.1. Amaya

Amaya [Quint 05] is an authoring tool for the web. Its main role is to help authors to easily create, edit and publish documents that comply with the standards of the web and exploit most of their features. More precisely, it allows authors to interactively manipulate the structure of XHTML documents [Altheim 01] through a formatted representation. It does so following the XHTML DTD: all user actions are performed under the control of the DTD. As a consequence, documents produced by Amaya are well-formed and valid (in the XML sense) and the user does not have to worry about what is allowed where, or about the XML syntax.

Amaya can also edit documents encoded in other XML languages, in the same way as XHTML pages, i.e. based on their DTD. It supports MathML [Carlisle 03] for mathematical expressions and SVG [Ferraiolo 03] for animated 2D graphics. Formulas and graphics may be included in XHTML documents and edited within these documents, thus allowing compound documents to be treated as easily as XHTML pages.

In addition to these XML languages Amaya supports CSS [Lie 05], the style language for XHTML and other XML documents. Users can create and modify CSS style sheets while they are editing documents. They can thus develop style sheets step by step, while checking the impact of each change.

For each XML language supported (XHTML, MathML, SVG), Amaya follows closely the corresponding DTD. As a consequence, the dialogue between an author and the tool is based on the structure and the elements defined by the DTDs. While this approach was proven very efficient for many kinds of web pages, in a more specific context, such as a community of practice, one can go further. Documents produced and exchanged in CoPs are often more structured than usual web pages. A richer document model is needed to efficiently help authors. For this purpose, we propose to include in Amaya the support for templates, based on the Semantic XHTML approach.

2.2. Templates and Semantic XHTML

Consider the structure of this report. First comes the front page which contains information about the project (number, acronym, full name, type, etc.), the title of the report, two dates, a list of keywords and the list of authors with their affiliation. The document body consists in an introduction and a sequence of sections with nested subsections. It is followed by a list of bibliographic entries, which themselves have a well defined structure. While some parts, such as the front matter, have a very rigid structure, some other parts are not strongly constrained. A section for instance must start with a heading, but it may contain paragraphs, bulleted lists, figures, examples, tables, etc., and the author is free to choose the appropriate elements and to organize them in any order.

The traditional approach for modelling such a document is to write an XML schema [Fallside 04] or a DTD [Bray 04a], that describes exactly the intended structure of a Report. Then, an XML editor has to be used for producing documents that are consistent with the model. Consistency (validity in XML-speak) is checked with a validating parser when the document is complete. Before publication or communication to others, documents have to be converted into (X)HTML, to make sure that everyone can read and browse it with the most common tool, a web browser. This is usually achieved by a transformation expressed in a language such as XSLT [Clark 99]. A specific transformation sheet has then to be developed.

This process is long, complex and costly. In addition to the authors, it involves several actors with various technical skills, in particular to create a schema and to implement the transformations from the source language to the publication language. When only a limited number of documents are expected to be created, the cost of this process is too high. Authors prefer to write directly (X)HTML documents ready for publication. They simplify the whole process, but they loose the advantages offered by XML. The issue is that XHTML does not seem to be rich enough to represent all the details of the structure mentioned above.

Looking closer at this issue, it appears that XHTML can actually do the job, in particular by exploiting the `class` attribute. This attribute gives a more precise type to elements that are otherwise a bit vague, like `div` (division) and `span`. For instance, all the information about the authors can be wrapped in a `div` with an attribute `class="authors"`. In this `div`, the `p` (paragraph) that contain information about an author can be assigned an attribute `class="author"`. To refine the structure further in this paragraph, the given name, the family name and the affiliation can be separated in different `span` elements, each with a different `class` attribute, clearly identifying its role in the structure.

This is the *Semantic XHTML* approach, also called microformat [Khare 06]. It consists in defining a rich structure (a Report or the information about a person) in terms of another, less specialized language (XHTML), by stating guidelines for using the lower level language. This approach has many advantages:

- Documents can be structured with semantically rich markup.
- No transformation is needed for displaying a document with a simple web browser: the document is encoded in plain XHTML, without any extension.
- The structure provides detailed information that can be exploited by CSS style sheets [Lie 05] to fine tune the style and the layout of documents.
- All the details of the structure are available when the document is exchanged. Any application can extract and reuse information from these documents.

Semantic XHTML has a number of advantages, but also some limitations. First, more markup is required than for dumb HTML. Producing it by hand is tedious and error-prone. Second, no formal specifications define these formats, and this is a problem. If the additional semantics are not correctly encoded in the XHTML markup, most of their benefits are lost: style sheets do not work correctly and applications can not retrieve the information they are supposed to process.

To address these issues we propose a language and an editing tool. The tool, based on Amaya, makes editing Semantic XHTML easier, simpler and more effective. The language, called XTiger (eXtensible Templates for Interactive Guided Edition of Resources), allows Semantic XHTML to be clearly described. The editing tool uses document descriptions expressed in this language to help authors produce valid documents, i.e. documents where the additional semantics are correctly encoded.

2.3. The XTiger language

The main role of the language is to describe a *generic structure* in terms of another structure representation language called the *target language*. The target language considered above is

XHTML, but it could be any other XML language, for instance MathML [Carlisle 03] for mathematical expressions, SVG [Ferraiolo 03] for animated graphics, etc. It could even be a mix of several XML languages, thus addressing compound documents. The generic structure is a model from which document instances are derived. All instances derived from the generic structure are supposed to comply with the model.

In this regard XTiger is very close to schema languages, and it borrows some of its features from these languages. But XTiger is much simpler than schema languages, as it relies on the target language. Thus, it does not need to define every single part of a document type. It only adds more constraints to an existing document type where it is needed.

XTiger is designed to be used in combination with a target language. It describes a generic structure under the form of a template. Templates exist in many HTML editors (Dreamweaver, FrontPage, etc.). Some have a very basic notion of a template. They just propose a series of typical documents that can be freely edited and adapted without any dedicated mechanism. Some others support a template language that is interpreted by the editor. In most cases this language is embedded in comments whose content must follow a specific syntax. In Dreamweaver, for instance, these special comments can prevent the user from modifying a part of the template document, they can allow him/her to repeat a given part (possibly 0 time), they can finally let some parts completely free. For XTiger, we need more expressive power. The template should provide a fine-grained mechanism for controlling the structure of document instances. It should be able to set different levels of constraints on the contents of a document.

In our approach, a template is a skeleton document, expressed in the target language, which contains elements of the XTiger language in various places. The role of these XTiger elements is to specify what elements and attributes of the target language must, should or could be present at these places, possibly with some predefined values. That is the core of the language, which specifies the structure and the content of document instances. It is complemented with other features that make the language easier to use. These features allow for instance structure fragments to be defined once and to be used at several places, in one or several templates. They facilitate a modular construction of document types, by sharing reusable pieces of structure.

XML is a natural choice for the syntax of XTiger, as its main role is to describe structures. In addition, using the same syntax for the structure description language and for the target language can simplify implementation. XML also provides the namespaces mechanism [Bray 04b] for mixing different languages while still making a clear distinction between them. This is also a distinctive feature. In many HTML editors the template language is hidden in comments and has no structure.

Taking advantage of XML namespaces, in the following we use the prefix `t:` for all names from the XTiger namespace, while names from the target language are not prefixed.

2.3.1. Types

Like in a schema language, types can be defined in XTiger. Types are used to define pieces of structure that may occur at several places in a template or in several templates.

Types are built from basic types using constructors.

In the initial version of XTiger, there are a few basic types, like in programming languages: number, string, boolean. In the future this short list could grow and include datatypes like those used in XML

Schema [Fallside 04]. These basic types may be used to specify the content of some part of a document. For instance, the three `span` elements that appear in example 1 use the basic type `string` for their content (element `t:use` is presented in section 2.3.3 below).

Basic types are also used to build more complex types. Element `t:component` is a constructor that creates a new type containing different elements, both from the XTiger language and from the target language. In the Report template, we could define a type `refbook` that would be useful in the bibliography to refer to a book (see example 1 for a simplified version). We can similarly define other types of bibliographic citations to refer to articles (`refarticle`), technical reports (`refreport`), etc.

```
<t:component name="refbook">
  <p class="refbook">
    <t:repeat minOccurs ="1">
      <span class="bibauthor">
        <t:use types="string"/>
      </span>
    </t:repeat>
    <span class="title">
      <t:use types="string">Book title</t:use>
    </span>
    <span class="pub">
      <t:use types="string">Publisher</t:use>
    </span>
  </p>
</t:component>
```

Example 1: a component

Element `t:union` is another constructor. It defines a new type as the union of other types. In example 2 the new type `bibref` is defined as the union of the different types of bibliographic citations we have already defined.

```
<t:union name="bibref" include="refbook refarticle refreport"/>
```

Example 2: a union of types

To simplify the writing of templates, a few unions are predefined. Union `any` includes all basic types, plus all types defined by the `component` and `union` elements, plus all types of the target language. The other predefined unions are more selective: `anySimpleType` includes only the simple types, `anyComponent` includes only the types defined by a `t:component` element, `anyElement` includes only elements of the target language.

2.3.2. Structure description

In a template, everything specified in the target language must occur as is in all document instances, except when a XTiger element states otherwise. The top level structure of a Report like this one (title, authors, introduction, sections, bibliography) is represented in a template simply by its XHTML markup. The contents of these elements are obviously not pre-defined. They must then be represented by some XTiger elements.

One of these elements is `t:repeat`, which allows a structure to be repeated several times. The structure to be repeated is the content of the `t:repeat` element. Example 3 shows how the introduction of a Report is defined as a `div` that contains a `h2` (heading in XHTML) with a predefined content, followed by a sequence of `p` (paragraph) elements.

```
<div class="intro">
  <h2>Introduction</h2>
  <t:repeat>
    <t:use types="p"/>
  </t:repeat>
</div>
```

Example 3: the repeat element

In example 3, the structure to be repeated is a single element of type `p` in the target language, and the content of this element is completely free. In many cases the structure is not that simple. Types defined by a component or a union may be used in these cases. Example 4 shows how a more constrained structure can be defined. It specifies the structure of the bibliography in the Report template. After a `h2` element with a predefined content, five to twenty instances of the structure defined by type `bibref` (see example 2) may occur. The number of occurrences of the repeated structure is specified by two attributes, `minOccurs` and `maxOccurs`, which indicate respectively the minimum and maximum number of occurrences.

```
<h2>Bibliography</h2>
<t:repeat minOccurs="5" maxOccurs="30">
  <t:use types="bibref"/>
</t:repeat>
```

Example 4: bibliography template

A structure that can be repeated 0 or 1 time is treated as a special case, as authors consider this as an option rather than a repetition. It is represented by an element without any attribute, `t:option`.

2.3.3. Content and attributes

The basic types and the types defined by elements `t:component` and `t:union`, as well as the predefined unions, are used by the `t:use` element. Wherever this element appears in a template, it indicates what type of element can be inserted at this position. This includes also the types from the target language.

The `t:use` element constrains the type of the element that can appear at its position, but not their contents. In example 3, the elements following the `h2` must be paragraphs (`p`), but these paragraphs may contain anything. Note that in example 1, two `t:use` elements have a content. This is considered as an indication of a possible content, but it may be changed freely in a document instance. This is different from the content that appear in a target language element, which can not be changed in an instance document: the content of the `h2` in example 3 must always be "Introduction".

Obviously, it is possible to restrict the content of an element produced by `t:use`. If the `types` attribute of a `t:use` specifies a component, the `t:use` element must be replaced in a document instance by the exact structure of the component. Only the sub-elements of the component that are explicitly not fixed can be different. According to example 1, an element `<t:use`

`types="refbook">` will be replaced in an instance by a `p` element with an attribute `class="refbook"` and this element will contain one or more `span` elements with attribute `class="bibauthor"` followed by a `span` with `class="title"` and another with `class="pub"`. Only the number of authors and the content of the span elements are free.

While `t:use` is mainly used to constrain the structure, the `t:bag` element brings some freedom where it is needed. It defines the set of types that can be used in the subtree that will replace it in a document instance. It constrains only the element types, but not the way they are assembled. The `t:bag` element has a `types` attribute which specifies the allowed element types. This is a list of type names that may contain basic types, unions, components, and element types from the target language. In the Report template, we use `t:bag` to define the content of a section. See example 5 which defines the body of a Report.

```
<t:repeat minOccurs="1">
  <div class="section">
    <h2>
      <t:use types="string">Heading</t:use>
    </h2>
    <t:repeat>
      <t:bag types="anyElement"/>
    </t:repeat>
  </div>
<t:repeat>
```

Example 5: a bag

With this definition, any XHTML elements could appear after the initial `h2` in a section. In fact, the definition is a bit more restrictive. Like all other XTiger elements, `t:bag` is not supposed to violate the structure of the target language. XTiger simply adds more rules for using the target language. The rules stated in the DTD or schema of the target language still apply. In the section defined in example 5, although the XTiger definition of `anyElement` includes XHTML elements such as `head`, `body`, or `meta`, the XHTML DTD makes them invalid within the `div` element representing a section.

XTiger does not consider only the elements of the target language, but also its attributes. Element `t:attribute` defines rules for using an attribute from the target language. It always appears as a child of an element of the target language and specifies how to use a given attribute with this element. It may make an attribute mandatory or optional, it may specify a fixed value, a default value or let the value free. It may also prohibit the attribute to be associated with the element. Again, these constraints should not contradict those from the DTD or schema.

```
<t:component name="author">
  <p class="author">
    <t:option>
      <img class="portrait">
         <t:attribute name="alt" default="picture of the author"/>
         <t:attribute name="width" use="optional"/>
      </img>
    </t:option>
  ...
</t:component>
```

Example 6: defining attributes

Example 6 defines the type author for use in the Report template. This type includes an optional element `img` which is supposed to be a picture of the author. The first `t:attribute` in the example makes the `alt` attribute of the `img` element mandatory and provides a default value for it. The second `t:attribute` element states that the `img` could have a `width` attribute. Note that the XHTML DTD already says so, but if this statement was not present, users could not set the width of these images: attributes that are not explicitly mentioned in a template are forbidden in the instance. Nothing is said about the `src` attribute of the `img`, but the XHTML DTD makes this attribute mandatory. It is then mandatory in any instance derived from the template where the component `author` is used. There is simply no additional constraint on this attribute in type `author`.

2.3.4. Other features

Elements `t:component` and `t:union` define new types. In a template, they are grouped at the beginning of the document in a special element, `t:head`. This element is unique in the template and must appear before using any type it defines.

In addition to type definitions the `t:head` element may contain some `t:import` elements, which import external definitions for use in the template. These external definitions are grouped in separate documents called *libraries* which, like element `t:head`, contain type definitions and `t:import` elements: a library may import other libraries.

Element `t:import` has a single attribute, `src`, that contains the URI of the library to be included. The order of the `t:import` elements is used to choose among different type definitions that have the same name: the latest imported library wins.

Libraries allow types that are used in different templates to be shared. They are especially useful for popular microformats. In the Report template, types `bibref`, `refbook`, `refarticle`, `refreport` would typically be defined in a library to be used also in templates for articles, theses, and other types of scholar documents.

2.4. Implementation issues

XTiger was designed to be implemented in a document editor. It represents generic structures in a way that could be efficiently used by an editor to help authors structure and encode Semantic XHTML. It has now to be implemented as an extension of Amaya. In this implementation work the main issues to be considered are structure manipulation and user interface. In this report we only mention the main directions for addressing these issues.

2.4.1. Structure manipulation

In its original version, Amaya provides the editing feature for the target language. For supporting XTiger templates, Amaya has to be extended to handle the generic structure defined by the XTiger elements, in addition to the XHTML structure. These two structures are tightly mixed, and the editor has to handle them simultaneously. Fortunately, Amaya already supports namespaces. But the XTiger namespace is a bit special as compared to other namespaces that are already supported in Amaya. It is not a document format, like SVG or MathML, for instance. The XTiger elements are not supposed to be displayed as part of the document, but their semantics must be interpreted by the editor to guide the editing process and to make sure that the editing actions of the user are executed in accordance with the generic structure described by the XTiger template.

When a user wants to create a document from a template, the new document instance is created as a copy of the template. The `t:head` element with its type definitions, however, is not copied in the document instance, as type definitions are intended to be shared. The template is linked to the new instance by a processing instruction inserted at the beginning of the instance, in the same way CSS style sheets are linked to an XML document. With this link, the editor will find all the type definitions needed during the subsequent editing sessions. All other XTiger elements (`t:use`, `t:bag`, `t:repeat`, `t:option`, `t:attribute`) as well as all target language elements are kept in the copy that constitutes the initial instance. XTiger types that appear in these elements are replaced by references to their definition in the template (actually, by references to a more compact and more accessible representation of types in core memory).

When the instance has been created, the user can see a skeleton of the document to be written. He/she then interacts with the editor to develop the structure and provide contents. This is done following all constraints, those expressed by the XTiger elements and those of the target language expressed by its DTD. When creating new elements, both target language elements and XTiger elements are created. For instance, when adding a new element in the `t:repeat` that allows several sections to be created in a Report, the whole template of the new section is created (see example 5). This allows the editor to know what is allowed by the template and what is not, just by looking at the current position in the document: the local XTiger elements express the local constraints. As a consequence, the edited document can be seen as a template that grows.

Guiding the user and checking the document structure is basically what Amaya is natively doing for the target language, but the difference with templates is that the generic structure described by XTiger elements is not separated from the document, like a DTD or a schema. It is part of it. There is no need to first go to the DTD or schema and find the rules that are relevant to the current position. On the other hand, the XTiger elements and attributes that are interspersed in the document have to be processed in a special way. When editing a document instance, they are not supposed to be modified by the user in the same way a mathematical expression can be modified within a XHTML page for instance.

As an example, consider how a new section may be added in a Report. According to example 5, the editor is allowed to create a section as a child of the `t:repeat` element. Following the definition of example 5, it creates this structure:

```
<div class="section">
  <h2><t:use types="string">Heading</t:use></h2>
  <t:bag types="anyElement">

  </t:bag>
</div>
```

Notice that it repeats the `t:bag` and `t:use` elements from the definition, thus developing the template and indicating what is allowed at each position. The content of the `t:use` element is an ordinary string, "Heading", that can be edited freely. However, this string is still enclosed in a `t:use` element that prevents the editor from adding any other type of element in the `h2`.

The generated `t:bag` element tells the editor that it is allowed to create any element of the target language in it, at the position of the empty line. If this element was not generated, the editor could not create anything there, in the same way it can not create anything before the `h2`. The generated structure for the new section is considered as part of the template, and then it can not be modified where it is not explicitly stated, like in the `t:use` or `t:bag` elements.

2.4.2. User interface issues

Even if XTiger elements are not supposed to be displayed like elements from the target language, it is useful for the user to see them, in order to know what is allowed at each position in the document. For this purpose, a special visual representation is needed. Areas where the user can make changes will be displayed with special borders. Other parts of the document are locked: editing is not allowed there.

The user needs not only to visualize the XTiger elements, but also to interact with them. He/she needs to create or remove elements in a `t:repeat`, to create or delete the content of a `t:option`, to insert the content of a `t:use`, to create elements in a `t:bag`, and to edit attributes according to a `t:attribute`. When creating the content of these XTiger elements, choices have to be made among the possibilities offered by a `t:union`.

Some of these operations can be done with the usual user interface of Amaya. For instance, deleting an element in a `t:repeat` will be done with the usual Delete command (however, the Delete command has to be extended to avoid deleting elements that are made mandatory by their parent XTiger element). The user interface of Amaya will be extended to allow the user to perform all these operations.

3. Authoring model for LimSee3

Amaya is dedicated to compound documents. It can handle documents containing different forms of information: structured text and images in XHTML, structured graphics in SVG, mathematical expressions in MathML. It handles only discrete media. For editing continuous media such as video, sound, animations, a different tool is available, LimSee2, which can produce document in the SMIL format [Bulterman 05]. SMIL is an XML language for multimedia documents where time and synchronization play a key role. It is sometimes called the XHTML of multimedia.

LimSee2 is based on the same principle as Amaya. It is controlled by the SMIL DTD and helps authors to produce valid SMIL documents. Like Amaya, its authoring model is very close to the format it handles and for specialized documents a more powerful model is needed.

3.1. From LimSee2 to LimSee3

When developing LimSee2 the goal was to provide graphical UI components to assist the user in the edition of a SMIL document. LimSee2 is an open-source and cross-platform Java application that totally fulfills this objective. Nevertheless, this approach of directly manipulating the SMIL format might not appeal to the general user for several reasons:

- the intrinsic complexity of SMIL can not be ignored and a good knowledge of the SMIL language is required.
- the manipulation of the low level SMIL structures is not suited to complex authoring tasks.
- SMIL does not provide very rich information about the semantic structure of the document. The only way to give hints on the logical structure is to set meaningful `id` values to the SMIL structures.

- the extension of the SMIL format with foreign namespaces and/or metadata, while keeping a valid document, would quickly raise scalability and maintainability issues, along with the usual problems related to players compatibility.

Moreover, the design and architecture of LimSee2 might be improved. The application lacks a "pure" document model module. It partly mixes core information with UI components and it makes heavy use of expansive string manipulations (for parsing XML attributes values for instance).

This is why the idea of using an intermediate and dedicated authoring format has emerged. It would additionally offer the possibility to be exported to virtually any format from the SMIL family: SMIL itself (any version or any profile), XHTML+SMIL (or "HTML+TIME", effectively), Animated SVG, XMT-O, etc.

This intermediate format is the core of LimSee3, a successor of LimSee2 but a totally new software. The principles guiding the design of this new project are:

- Ease of use: the authoring format should be easily manipulated and the user interface should be as intuitive as possible.
- Expressivity: the authoring format should allow the creation of a wide set of complex hypermedia presentations.
- Extensibility: the authoring format should be effectively extensible, enabling complex authoring tasks as new functionality.
- Customizability: the application should be configurable by lots of user-defined settings; the interface should also be customizable.

3.2. Document model

While the LimSee2 editor was built to allow authors to manipulate all aspects of the SMIL language, LimSee3 will be less dependent on the language. It will still be able to generate documents coded in SMIL and other languages based on (or derived from) SMIL, as explained above, but it will not require the user to "think" in SMIL. It will present the user with a more friendly document model, where s/he will be more comfortable, manipulating entities that are closer to what s/he has in mind in terms of document model.

3.2.1. Object approach

From the user perspective, a multimedia document is logically structured in several high level objects that have their own semantics:

- The *slide show* is composed by a *sequence of images* and a *navigation bar*.
- The *navigation bar* has two buttons, a *previous button* and a *next button*.
- A *navigation button* is an *image* with a *link* to another object.

This object approach has been chosen for the LimSee3 authoring format: a LimSee3 document is a nested structure of semantically significant *objects*. This strong semantical cohesion is very important in terms of usability: objects can easily be abstracted and manipulated as entities by the user.

Note: In SMIL, the definition of these high-level objects is distributed in different parts of the document (in the header for the spatial information, in the body for the temporal information).

Each object defines its own *temporal* and *spatial* dimensions. The object may specify how it can interact with other objects if needed. Additionally, dedicated edition services may be associated to each object according to their specific semantics.

The whole document can then be viewed as a *tree of objects*, as shown in the following example:

```
+ slideshow
  + sequence of slides
    + slide
      - image
      - title
    + slide
      - image
      - title
  + navigation bar
    + previous button
      - image
      - link
    + next button
      - image
      - link
```

3.2.2. Objects hierarchy

3.2.2.1. Media assets and media objects

A *media asset* is a reference to an external resource which is totally independent of any integration context. It is referenced consistently by its URI. It can be for instance an audio track (an mp3 file, a real audio stream, etc.), a video, a piece of text, an image, an animation, etc. A media asset is not an *object* as specified.

Each media asset has at least two properties: its *media type*, which specifies the family of media to which it belongs (image, audio, video, text, animation), and its *media encoding*, which specifies the format used to store its content (for instance jpeg, wav, mov, wmv). The media type and media encoding properties can be determined from the underlying resource by either its MIME type or the usual file naming conventions (file extensions).

A *media object* is the simplest object (i.e. the *atomic* object) of the authoring format. It is a wrapper around a media asset, but it also defines the properties related to its integration in the document (mostly spatial and temporal information). A media object can optionally apply minor transformations to the wrapped media asset (cropping, transparency).

It is very similar to the SMIL media object element:

```
<img id="firstImage" src="images/image_1.jpg" fill="freeze" begin="0s" dur="10s"/>
```

Note: The `src` attribute is identifying the media asset wrapped by this media object.

3.2.2.2. Rich objects

A *rich object* is a composite object, a complex structure of other nested objects. The XML element containing the definition of a rich object is called `object`.

Each rich object is defined by at least the following information:

1. The list of its children objects, i.e. the sub-objects that are nested in this object. This is the `children` element.
2. A section that describes the timing scenario of this object, i.e. the information that describes the synchronization of its internals. This is the `timing` element.
3. A section that describes the spatial layout of this object, i.e. how the internals of this object are positioned in the space allowed to this object. This is the `layout` element.

In XML, the global structure of a rich object is hence the following:

```
<object>
  <children>
    <!-- a list of the sub-objects -->
  </children>
  <timing>
    <!-- the internal timing scenario -->
  </timing>
  <layout>
    <!-- the internal spatial layout -->
  </layout>
</object>
```

The timing section can contain SMIL timing elements, such as `par`, `seq` or `excl`, which should be in the SMIL namespace. Sub-objects of this object can be used in the timing section referenced by their ID with the `timeRef` element, as in the following example:

```
<timing xmlns:smil="http://www.w3.org/2001/SMIL20/">
  <smil:par>
    <timeRef refId="subObj1" smil:begin="0s" smil:fill="freeze"/>
    <timeRef refId="subObj2" smil:begin="1s" />
  </smil:par>
</timing>
```

The layout section can contain SMIL `region` elements which should be in the SMIL namespace. Sub-objects of this object can be used in the layout section referenced by their ID with the `layoutRef` element, as in the following example:

```
<layout xmlns:smil="http://www.w3.org/2001/SMIL20/">
  <smil:region id="objRegion" left="0" top="0" height="100" width="50">
    <layoutRef refId="subObj1" smil:height="50" smil:top="0"/>
    <smil:region id="subObjRegion" height="50" top="50"/>
  </smil:region>
</layout>
```

3.2.3. Object-to-object relations

In the LimSee3 authoring format, an object is an almost independent entity. However, in most cases, these objects will be related one with another. For instance in a slideshow the navigation bar is somehow related with the sequence of slides, as previous and next buttons are actually links to slide objects.

To represent these object-to-object relations, an object can contain a dedicated `related` element. This element declares symbolic references to external objects, which can then be used in the timing and layout sections of the master object.

An object can have access to the information of a related object by using an XPath expression. This information can be used for instance to set attribute values in the layout and timing section using an XSLT-like syntax.

In the following example, the object `myObject` is a sequence of two audio tracks that starts when the external object `extObject` ends:

```
<object id="myObject">
  <related>
    <ref name="master" refId="extObject"/>
  </related>
  ...
  <timing xmlns:smil="http://www.w3.org/2001/SMIL20/">
    <smil:seq>
      <attribute name="begin">
        <value-of select="@id" refName="master"/>.end
      </attribute>
      <audio src="audio1.mp3"/>
      <audio src="audio2.mp3"/>
    </smil:seq>
  </timing>
  ...
</object>
```

3.2.4. Comments on the document model

By defining almost independent objects, the LimSee3 authoring format focuses on the logical structure of the multimedia document. It is actually a kind of reorganization of a SMIL document in order to ease the manipulation of high-level entities with strong semantics. This may greatly facilitate complex authoring tasks such as the insertion of a whole complex object in a multimedia document as a single entity, performed for instance by a unique simple drag-and-drop operation.

The power of XML in terms of structural transformation and extraction of information allows the user to view the same document from different perspectives:

- the complete structure (the whole XML tree)
- the logical structure (the tree of objects)
- the timing scenario (global or per object)
- the global layout hierarchy (global or per object)
- the objects dependency graph

- the table of media assets used in the document

Having access to these different views of the document allows each authoring task to use the one that is the most appropriate.

3.3. The templates model

The user of an authoring tool may want to create several instances of a same class of document (for instance a slide show, a quiz, a captioned movie). To facilitate such redundant creation of complex documents, the authoring format permits the definition of *template documents*. These are special documents having a preset *shape* and used as starting point to create document instances.

3.3.1. Template documents and template objects

A *template object* is the abstraction of an object from any integration context. The definition of objects as entities facilitates this concept. Note that the declaration of related objects in a separate element and the use of mere symbolic references in the object body greatly facilitates this abstraction.

A *template document* is a document with a structure predefined by a set of constraints. The constraints on the document structure may be viewed as *reserved places* to be filled with instance-specific content and that may have incidence on other parts of the document. A template document is actually a document composed by objects and template objects.

3.3.2. Media zones

A *media zone* is the most basic template. It can be viewed as a reserved place for a media asset. It is actually a media object that does not yet reference a media asset, as in the following example:

```
<img id="firstImage" fill="freeze" begin="0s" dur="10s"/>
```

The application knows that a reference to a media asset (i.e. a URI) is required at the place where a media zone is inserted and then invites the user to fill the required information.

The template can optionally define a customizable invitational text that can be displayed to the user.

```
<img id="firstImage" fill="freeze" begin="0s" dur="10s">
  <invitText>Please add image here</invitText>
</img>
```

The invitational text will be removed once the zone is filled with a media asset.

3.3.3. Repeatable structures

A *repeatable structure* is an homogeneous list of objects, i.e. a list composed by objects the structure of which matches the structure of a specified model object. Such a list is declared with the `objList` element. The model for the children of a list is declared in the `model` element (the first child element) as in the following example:

```
<object id="SlideSequence">
    ...
  <children>
    <objList>
      <model>
        <object>
          <!-- the definition of a slide -->
        </object>
      </model>
    </objList>
  </children>
    ...
</object>
```

The addition of an item to the list of objects is a simple copy operation of the model object to the list content.

Note that the cardinality of the list may be specified by minimum and maximum item counts.

The items of a list of objects can be used in the timing and layout section of the parent object with the `timeRefList` and `layoutRefList` elements. Note that the list can select only a subset of the list items if required, with an XPath expression.

```
<object id="SlideSequence">

  <children>
    <objList id="slides">
      <!-- a list of slides -->
    </objList>
  </children>

  <timing>
    <seq>
      <timeRefList refId="slides" />
    </seq>
  </timing>

  <layout>
    <region id="regionSlides" height="400" width="200">
      <layoutRefList refId="slides"/>
    </region>
  </layout>

</object>
```

Note that since such a list of objects is homogeneous and defines a model for its children, the structure of each child is well known and can be accessed finely with XPath to extract useful information. This information can then be used to set attribute values in the layout and timing section with an XSLT-like syntax. This is a powerful mechanism for defining structures that depend dynamically on the list content.

Suppose for instance we want to create a hierarchical list of entries and the hierarchical level is declared in a `param` element with the name `level`. The layout section of this list of entries could be:

```
<layout>
  <layoutRefList refId="Entries">
    <attribute name="left">
      <number value="10*number(param[@name='level']/@value)"/>
    </attribute>
    <attribute name="top">
      <number value="20*count(preceding-sibling::*)"/>
    </attribute>
  </layoutRefList>
</layout>
```

3.3.4. Instantiation of a template

The concepts of objects and templates are very similar. Actually, a template is an object that is not *complete*, lacking several parts of information: references to related objects, list items, references to media assets, etc. The *instantiation* of a template is the operation of providing all this lacking and required information. A template is instantiated when:

- Each media zone has been filled with a media asset.
- Each object list has been filled with object items (possibly 0).
- Each reference to a related external object has been resolved.

3.4. Issues and future extensions

3.4.1. Locked parts

The authoring format, in particular the template language, does not prevent the user to modify parts of the document. Templates are mere *guides* to the edition of a document, but the user can still modify any part of the document if s/he wants to.

It could be useful to allow the definition of *locked* parts in a template, i.e. parts where the user is not allowed to modify anything. This could be useful for instance to guide more strongly inexperienced users by restricting their access to the only parts of the document that make sense to them.

3.4.2. Heterogeneous lists

Repeatable structures are defined as homogeneous lists of objects. However, it may be useful to allow multiple types of content in such a list. For instance the user might want to define a list of either images or text objects. The trouble is that when dealing with heterogeneity the structure of each list element is unknown.

3.4.3. Object types

The objects of this document model are not *typed*. It could be very useful to specify that a given object conforms to a specific type, for instance a "sequence of images" or a "slide". Moreover, such types give precise knowledge about the structure of objects, which can result in finer usage of external references for instance.

However some issues remain unsolved, as the design of the inheritance of types, or the inference of a type given a specific object. Given the complexity of dealing with strongly typed structures (modification, validation) and defining such types, the authoring format does not follow this approach.

An intermediate step is to define only a few *classic* patterns. For instance a `sequence` type would be an object which is composed by a single object list and whose timing section would consist in a single `seq` element containing references to the list items.

3.4.4. High level concepts and syntax choices

Sometimes there are several possibilities to express the same idea. For instance, in SMIL a synchronization relation between two lists of objects could be described by syncbase begin values or by multiple `par` elements in a `seq` parent.

However, from the user's point of view the result is the same. The authoring format should choose a consistent syntax and optionally allow to use other solutions at export time.

Among the high level concepts that the authoring format might define are:

- *synchronization* relations
- the *state* of an object
- the *style* of an object

4. Document reuse tool

The aim of the information reuse tool is to allow existing documents initially structured for a given purpose to be restructured for use in a different context, thus improving reusability and information sharing between communities of practice. A notion closely tied with structured document reuse is that of structure transformations. Schema matching is a critical step in structured document transformations. Manual matching is expensive and error-prone. It is therefore important to develop techniques to automate the matching process and thus the transformation process. The information reuse tool takes two XML schemas as input and produces a mapping describing the similarities between those schemas. The latter output serves as the basis for the automatic generation of transformation scripts.

4.1. Automating structured document transformations

4.1.1. Related work

Several approaches focusing on automating XML document transformations have been recently proposed in the document community. Examples include the work described in [Leinonen 03], where authors propose a syntax directed approach for automating structure transformations between two grammars based on finite state tree transducers. This approach presents several limitations. First it works only if the two grammars have common parts, which restricts the scope of transformations to local transformations. Moreover, this approach is unable to resolve the heterogeneity that may occur between structured documents.

The authors of [Su 01] propose an approach for automating the transformations of XML documents. To this end they define a set of DTD transformation operations that establish the semantic relationships between two DTDs. The approach is based on a tree matching algorithm, called *DMatch*. The matching process is based on both provided auxiliary semantic information and a cost model. This approach presents several limitations. First the matching algorithm used is only able to discover one-to-one correspondences between DTDs and does not deal with many-to-many matches. Second, the matching algorithm requires additional semantic information to work correctly, which limits the scope of its application, since such semantic information is not always available. Finally, the matching algorithm used is inspired by work done in the area of tree matching and is unable to deal with the current XML schema model.

Additionally to the document community, database and artificial intelligence communities have widely considered the schema matching problem in many application domains. With the growing use of XML, several matching algorithms take into consideration the hierarchical structure of XML. In the following, we present some examples and check their applicability in the context of XML data transformations.

Cupid (Microsoft Research)

Cupid is a hybrid matcher combining several matching methods [Madhavan 01]. Cupid transforms the original XML schemas into trees and then performs a bottom-up structure matching. The basic assumption behind the structure matching phase of Cupid is that much of the information content is represented in leaves and that leaves have less variation between schemas than internal structures. Thus the similarity of inter-nodes is based on the similarity of their leaf sets. Schema structure in Cupid is used as a matching constraint, that is, the more the structures of the two nodes are similar, the more the two nodes are similar. For this reason, Cupid faces problems in the cases of equivalent concepts occurring in completely different structures, and completely independent concepts belonging to isomorphic structures.

Similarity Flooding (Stanford Univ. and Univ. of Leipzig)

In [Melnik 02], authors present a structure matching algorithm called Similarity Flooding (SF). For computing structural similarities, SF relies on the intuition that nodes of two distinct graphs are similar when their adjacent nodes are similar. The spreading of similarities in the matched models is reminiscent to the way how IP packets flood the network in broadcast communication. An iterative process is used to propagate similarities between nodes, where in every iteration the similarity of a map pair is incremented by the similarity of its neighbours. An important assumption behind the algorithm is that adjacency contributes to similarity propagation. Thus, the algorithm will perform unexpectedly in cases when adjacency information is not preserved. Furthermore, SF ignores all type of constraints while performing structural matching. Constraints like typing and integrity constraints are used at the end of the process to filter mapping pairs with the help of the user.

4.1.2. Shortcomings and requirements

As explained in the previous section, the proposed structural matching methods suffer from several serious shortcomings when applied in the context of XML documents transformations. The two basic problems we faced when trying to apply existent schema matching algorithms in the context of document transformations could be summarized as follows:

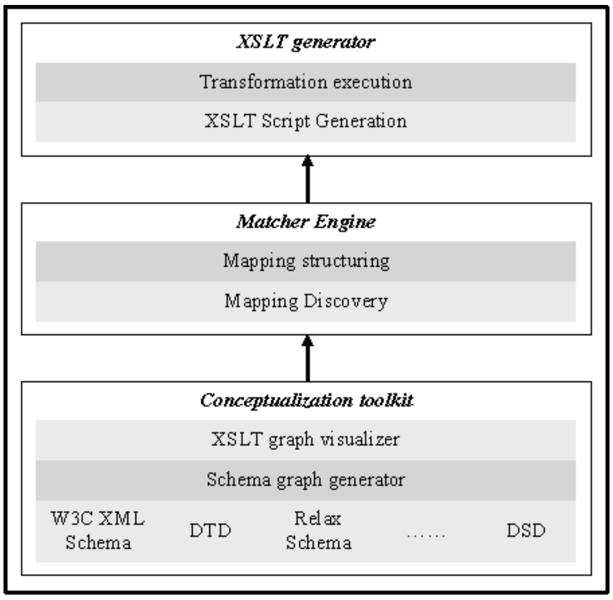


Figure 1: Prototype system

- Structural matching methods remain insufficient and very limited (they generally deal only with DTDs and exploit few structural characteristics, essentially parent-child relationships). Convinced that the structural organization in XML documents inferred some semantics of the data and traduced the designer point of view, a solution to XML schema matching problem *should exploit this information in a manner that increases matching accuracy.*
- The second fundamental problem concerns the result of the mapping process itself. Generally current schema matching algorithms only focus on discovering 1-1 mappings, also called "direct mapping". The result is a confidence score (ranging in [0,1]) between schemas' elements. Such a result is insufficient to perform transformations. *The proposed solution should discover "complex mappings"* (involving more than one source and/or target elements). In order to generate a transformation script, *the solution should be able to further discover transformation operations.*

4.2. Current state of the information reuse tool

The current prototype of the information reuse tool we have developed consists of three modules: *Conceptualization toolkit, Matcher engine,* and *XSLT generator* describing the phases that we consider fundamental in our work (Figure 1).

4.2.1. The conceptualization toolkit

The conceptualization toolkit consists on two modules: The schema graph generator and the schema graph visualizer.

4.2.1.1. The schema graph generator

As mentioned above, up to now few existent XML schema matching algorithms focus on structural matching exploiting all XML schema features. We propose an abstract model that serves as a foundation to represent conceptually W3C XML schemas and potentially other schema languages. We model XML schemas as a directed labelled graph with constraint sets, called *schema graph*. The schema graph serves to make clear XML schema features used within the matching process. In addition, it can be used in order to normalize XML schema languages into a uniform representation, hiding syntax differences and making structural and semantic heterogeneity more apparent.

Schema graph nodes

We categorize nodes into *atomic nodes* and *complex nodes*. Atomic nodes have no edges emanating from them. They are the leaf nodes in the schema graph. Complex nodes are the internal nodes in the schema graph. Each atomic node has a *simple content*, which is either an *atomic value* from the domain of basic data types (e.g., string, integer, date, etc.); or a *constructional value*, meaning a list value or a value or a *union value*. The content of a complex node, called *complex content*, refers to some other nodes through directed labelled edges.

Schema graph edges

Each edge in the schema graph links two nodes capturing the structural aspects of XML schemas. We distinguish three kinds of edges: (1) *containment relationship*, that is a composite relationship in which a composite node (“whole”) consists of some component nodes (“parts”); (2) *of-property* relationship, that specifies the subsidiary attribute of a node; and (3) *association relationship*, that is a structural relationship, specifying that both nodes are conceptually at the same level. Association relationships essentially model key/keyref and substitution group mechanisms.

Schema graph constraints

Different constraints can be specified with the XML Schema language. These constraints can be defined over both nodes and edges. Typical constraints over an edge are cardinality constraints. We also distinguish three kinds of constraints over a set of edges: (1) *ordered composition*, defined for a set of containment relationships and used for modelling XML Schema “sequences” and “all” mechanisms; (2) *exclusive disjunction*, used for modelling the XML Schema “choice” and applied to containment edges; and (3) *referential constraint*, used to model XML schema referential constraints. Referential constraints are applied to association edges, and are generally modelled through a join predicate. Other constraints are furthermore defined over nodes. Examples include *uniqueness* and *domain constraints*. Domain constraints are very broad. They essentially concern the content of atomic nodes. They can restrict the legal range of numerical values by giving the maximal/minimal values; they can limit the length of string values, or constrain the patterns of string values.

4.2.1.2. The schema graph visualizer

Generated schema graphs are displayed graphically using the schema graph visualizer (Figure 2). Such a graphical representation has three major advantages. First, it helps the user (which is not necessary familiar with the syntax of the XML schema language) to understand what both source and target schemas describe. Second, since we assume that the user knows the semantics of the target schema, based on such a graphical representation he can either add meta-information that

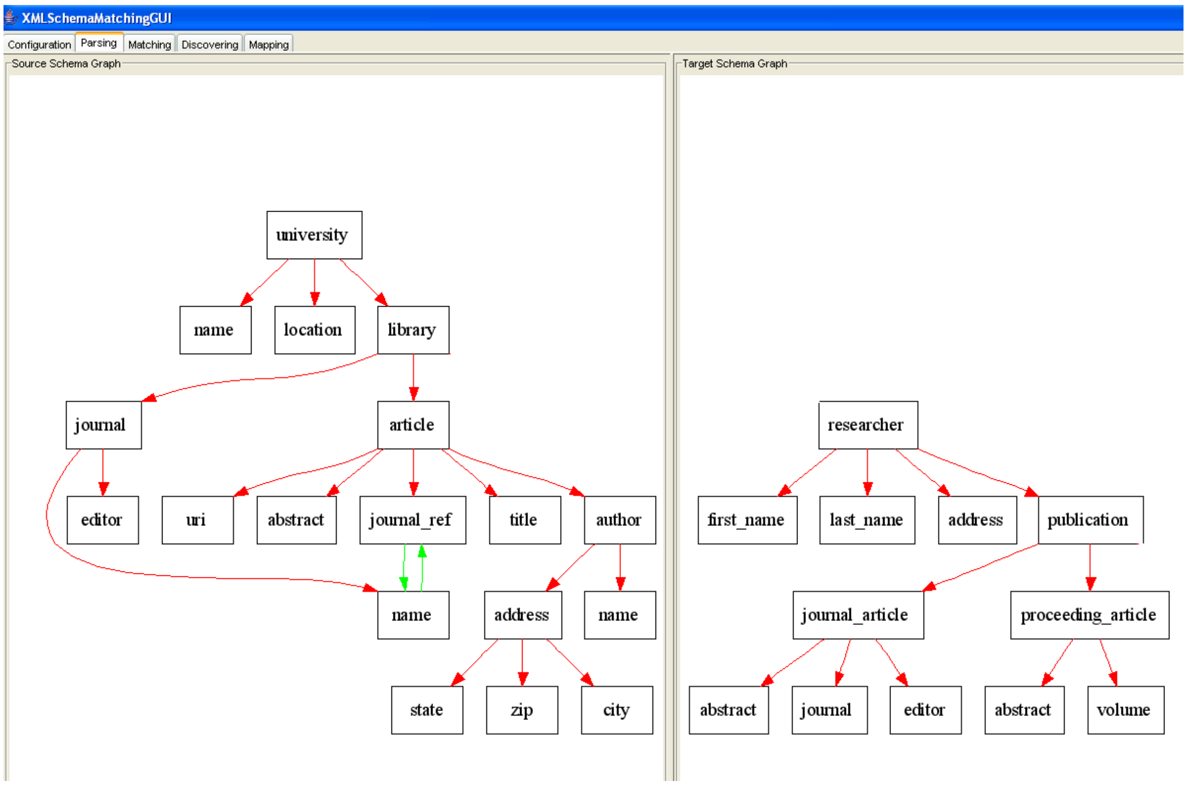


Figure 2: Schema graph visualizer

could help the matching process or even establish an initial mapping. Finally, this graphical representation could be used later in the mapping validation phase, especially when the user is invited to change the target schema (relax some constraints in order to make data reuse possible). Schema changes could then be done graphically without dealing with the XML schema language syntax.

4.2.2. The matcher engine

The matcher engine consists of two modules: *mapping discovery* and *mapping structuring*. To match schema graphs, we make use of four basic matching criteria (1) terminological matching, (2) datatype compatibility, (3) designer type hierarchy and (4) structural matching.

4.2.2.1. Terminological matching

The aim of this phase is to compute the similarity between schema nodes based on the similarity of their labels. To perform terminological matching, we make explicit the meaning of element names and establish semantic relationships between them based on WordNet. Our terminological matching is inspired essentially from Hirst and St-Onge's work [Hirst 98]. The idea behind Hirst and St-Onge's measure of semantic relatedness is that two concepts are semantically close if their WordNet synsets are connected by a path that is not too long and that does not change direction too often. A set of allowable paths have been then defined. The terminological similarity between two words is computed based on the length of the path relating them. Moreover, we identified four kinds of semantic relations between words, namely *equivalent* (≡), *broader than* (⊇), *narrower than* (⊆), and *related to* (~). The detailed algorithm is given in [Boukottaya 04] and [Boukottaya 05].

4.2.2.2. Datatype compatibility

To compute datatype similarities, we make use of the built-in XML schema datatypes hierarchy. XML schema datatypes are classified in multiple categories (called primitive datatypes) including for example Duration, Boolean, String, Decimal, etc. Each category has several derived datatypes. Two datatypes are considered to be similar if they belong to the same datatype category, and their datatype compatibility depends on their respective position in the XML schema datatype hierarchy. Based on the XML Schema datatype hierarchy, we construct a datatype compatibility table that gives a similarity coefficient between two given datatypes.

4.2.2.3. Designer type hierarchy

XML schema features concerning sub-typing, abstract types and substitution group mechanisms traduce the designer point of view and are used as a set of meta-data to help the matching process to discover both direct and complex mappings. The result of this step is a set of direct and complex mappings (essentially involving Union/Selection operators). Such mappings are be kept or rejected using either structural matching techniques or user intervention. In the case where type hierarchy is not available, we also make use of semantic relationships discovered by the terminological matching to derive complex matches. However, we give the priority to the designer type hierarchy since it reflects the designer point of view. More examples and algorithms on how to derive match candidates based on type hierarchy are detailed in our previous work [Boukottaya 04].

4.2.2.4. Structural matching

The matching techniques described in the sections 4.2.2.1, 4.2.2.2 and 4.2.2.3 may provide incorrect match candidates. Structural matching is used to correct such match candidates based on their structural context and thus derive correct direct and complex matches. Structural matching relies on the notion of node context. We distinguish three kinds of node contexts depending on positions in the schema graph:

- *The ancestor-context* of a node *n* is defined as the path (going through containment edges) having *n* as its ending node and the root of the schema graph as its starting node.
- *The child-context* of a node includes its attributes (through of-property edges) and its immediate sub-elements (through containment edges). The child-context of a node reflects its basic structure and its local composition.
- *The leaf-context:* Leaves in XML documents represent the atomic data that the document describes. The leaf-context of a node *n* includes the leaves of the subtrees (composed by containment relationships) rooted at *n*.

The structured context of a node is defined as the union of its ancestor-context, its child-context and its leaf-context. Two nodes are structurally similar if they have similar contexts. To measure the structural similarity between two nodes, we compute respectively the similarity of their ancestor, child and leaf contexts. The idea behind our proposed solution is to represent each node context as a path and to then rely on a path resemblance measure to compare such contexts. To achieve this, we relax the strong matching notion frequently used in solving query answering problem and use algorithms from dynamic programming [Boukottaya 04] [Boukottaya 05]. The result of this phase is a structural similarity coefficient (between 0 and 1).

4.2.2.5. Mapping discovery

Most schema matching algorithms produce similarity scores between source and target schema nodes such as the ones we produce in section 4.2.2.4. Such results partially solve the problem. First, similarities between individual nodes are not enough to produce access paths for retrieving data from the available sources. Second, all the produced mappings are one-to-one mappings, complex mappings identified using the type hierarchy have to be incorporated in the matching result and further complex mappings have to be discovered. For this we proceed in four steps:

Step 1: Compatible nodes identification

While generating mapping elements, we apply a *top-down strategy*. At the top level, we establish correspondences between complex nodes of the target and source schemas. Similar complex nodes are called *compatible nodes.*

Step 2: Context generation for compatible nodes

After identifying compatible nodes, we proceed to construct a context for each compatible node (the notion of context here differs from the context we defined in section 4.2.2.4). By taking edges around a complex node into account, we cluster a set of nodes and edges with a complex node as a conceptual component in the schema graph.

Step 3: Node mappings generation

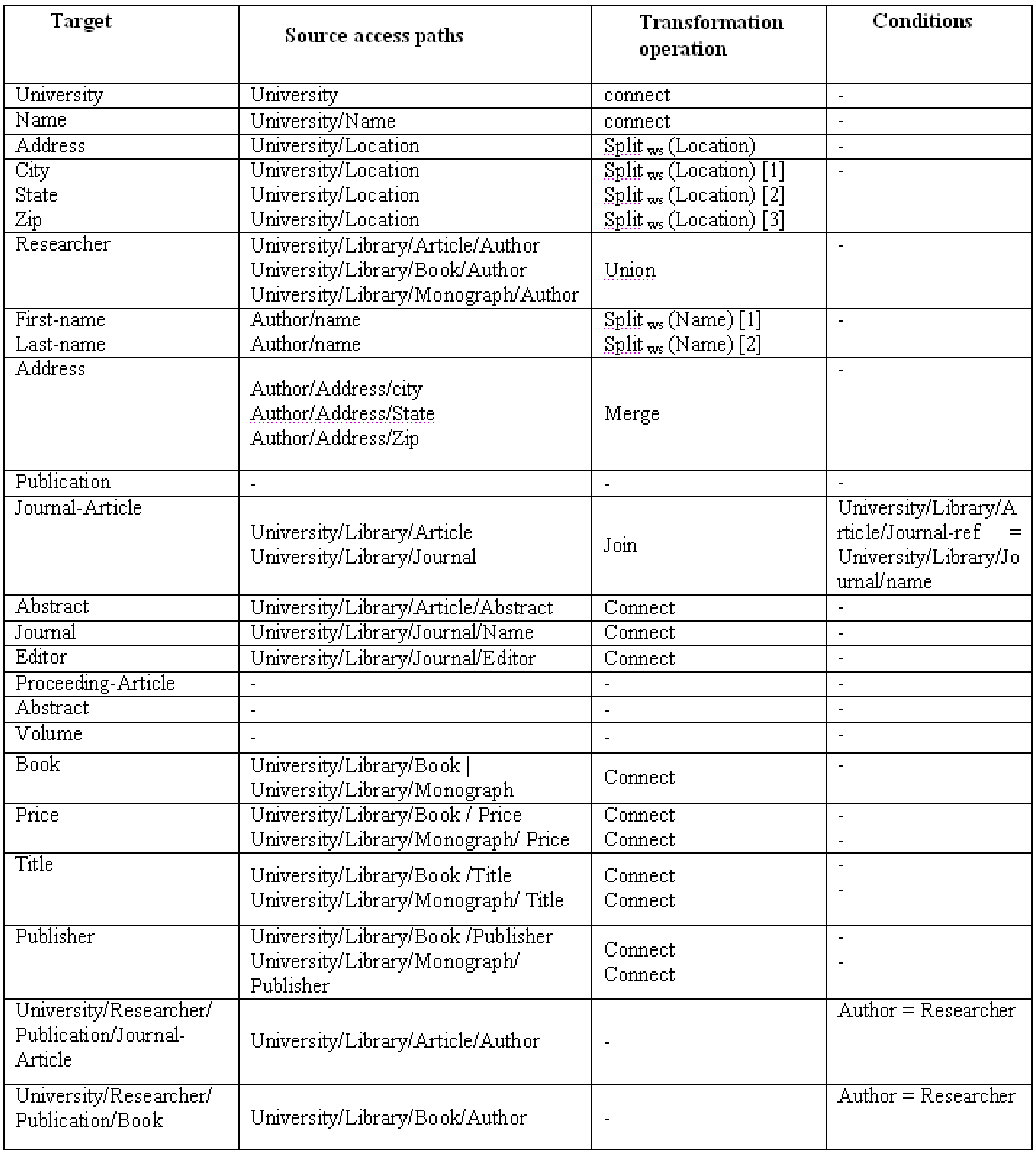
At this point, we finished with the top level comparison between source and target schema graphs. We are now ready to detect node and edges matches at the bottom level. For each matching pair which represents two compatible nodes in source and target schema graphs, we make use of node similarity score generated in section 4.2.2.4 to settle nodes matches.

Step 4: Access paths generation

We focus on the discovery of access paths in order to retrieve source data when performing transformations. For each target element, we first define the access path indicating where the matched source elements are localized, then the discovered transformation operation and finally the conditions under which the mapping element holds true. Examples of generated mapping rules are given in Figure 3.

4.2.2.6. Mapping structuring

After validation of the generated mapping rules by the user, we structure the mapping result using the XML Schema language and this for two reasons. First, it is easier to manipulate a structured mapping result either to modify it or to automatically generate transformation scripts. Second, structuring the mapping result greatly increases its reusability and adaptation, especially when schemas evolve. The nature of a mapping result may be understood by considering different dimensions, each describing one particular aspect: (1) *the entity dimension*, specifying schema entities involved in a mapping element; (2) *the cardinality dimension*: determining the cardinality of a mapping element ranging from direct mapping (1:1) to complex mapping (*m:n)*; (3) *the structural dimension,* reflecting how elementary mapping elements may be combined into more complex mapping elements; (4) *the transformation dimension,* reflecting how instances of the source schema are transformed during the mapping process; and (5) *the constraint dimension,* controlling the



Target	Source access paths	Transformation operation	Conditions
University	University	connect	-
Name	University/Name	connect	-
Address	University/Location	Split_{ws} (Location)	-
City State Zip	University/Location University/Location University/Location	Split_{ws} (Location) [1] Split_{ws} (Location) [2] Split_{ws} (Location) [3]	-
Researcher	University/Library/Article/Author University/Library/Book/Author University/Library/Monograph/Author	Union	-
First-name Last-name	Author/name Author/name	Split_{ws} (Name) [1] Split_{ws} (Name) [2]	-
Address	Author/Address/city Author/Address/State Author/Address/Zip	Merge	-
Publication	-	-	-
Journal-Article	University/Library/Article University/Library/Journal	Join	University/Library/Article/Journal-ref = University/Library/Journal/name
Abstract	University/Library/Article/Abstract	Connect	-
Journal	University/Library/Journal/Name	Connect	-
Editor	University/Library/Journal/Editor	Connect	-
Proceeding-Article	-	-	-
Abstract	-	-	-
Volume	-	-	-
Book	University/Library/Book | University/Library/Monograph	Connect	-
Price	University/Library/Book / Price University/Library/Monograph/ Price	Connect Connect	- -
Title	University/Library/Book /Title University/Library/Monograph/ Title	Connect Connect	- -
Publisher	University/Library/Book /Publisher University/Library/Monograph/ Publisher	Connect Connect	- -
University/Researcher/ Publication/Journal- Article	University/Library/Article/Author	-	Author = Researcher
University/Researcher/ Publication/Book	University/Library/Book/Author	-	Author = Researcher

Figure 3: Examples of generated mapping rules

execution of a mapping element. Based on the established mapping between the source and target schemas, a mapping generator relates the graphs of a source and a target schema by generating an instance of the mapping schema containing a set of mapping elements, each of which encapsulates all information needed to transform instances of source nodes into instances of target nodes. Figure 4 illustrates an example of mapping instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappingResult>
  <SourceSchema source="S1.xsd"/>
  <TargetSchema source="S2.xsd"/>
  <HasMappings type="OneToOneMapping" ID="map1"/>

  <mappingElement>
    <OneToOneMapping ID="map1">
      <Sourcenode name="University"/>
      <TargetNode name="University"/>
      <Transformation>
        <Operation name="connect"/>
      </Transformation>
      <HasMappings type="OneToOneMapping" ID="map1.1"/>
      <HasMappings type="OneToManyMapping" ID="map1.2"/>
      <HasMappings type="ManyToOneMapping" ID="map1.3"/>
    </OneToOneMapping>
  </mappingElement>

  <mappingElement>
    <OneToOneMapping ID="map1.1">
      <Sourcenode name="University/Name"/>
      <TargetNode name="Name"/>
      <Transformation>
        <Operation name="connect"/>
      </Transformation>
    </OneToOneMapping>
  </mappingElement>

  <mappingElement>
    <OneToManyMapping ID="map1.2">
      <Sourcenode name="University/Location"/>
      <TargetNode name="city"/>
      <TargetNode name="state"/>
      <TargetNode name="zip"/>
      <Transformation>
        <Operation name="split" param= "WS"/>
      </Transformation>
    </OneToManyMapping>
  </mappingElement>
  ....
</mappingResult>
```

Figure 4: Examples of structured mapping results

4.2.3. XSLT generator

An XSLT generator is designed to generate XSLT stylesheets from the structured mapping result. For each matching node pair, the XSLT generator traverses both the target schema graph and the mapping result in a depth-first manner and generates template rules. It permits the translation of data instances (XML files) valid against a source schema into instances valid against a target schema. The algorithm takes as input the target schema graph, the source and target instances, and the mapping result specification and produces an XSLT stylesheet consisting of a series of template rules, implementing the transformation. The XSLT generator proceeds in three steps:

Step 1: Initialize the translation

In this step, the XSLT generator tries to locate the first node of the target schema graph having a match candidate. For non mapped nodes, it acts as follow: if a non mapped target node is not mandatory (e.g., minOccurrence = 0, or is optional in the case of attribute nodes), nothing is generated, otherwise just element tags are generated with a warning message indicating that a value is needed to be added in order to ensure the validity (against the target schema) of the resulting instance. Once the first mapped node is localized, a queue is initialized and the template rules generation can begin.

Step 2: Traverse the target schema graph and the mapping result specification in a depth-first manner

In this step, the XSLT generator produces a construction template for the current node by using the current mapping element. For each sub-matching element, the XSLT generator adjusts the above template by inserting more construction or apply-template rules whenever necessary. For the case of atomic nodes, it inserts a new construction rule and no further process is needed for this node. Finally, the XSLT generator adds the adjusted templates into the final XSLT stylesheet. If a non mapped node is encountered, the XSLT generator acts as in step 1. If the queue is non-empty, the XSLT generator extracts one node as the current node and loops back to step 2, else it continues to step 3.

Step 3: Return the generated XSLT Stylesheet

In this step, the XSLT generator finalizes the generation of the transformation script. The structure of the target schema was respected throughout the process of XSLT generation, which guarantees the generation of valid target instances after the transformation process.

4.3. Extensions to the information reuse tool

With the work presented above, we have made a significant step into understanding and developing solutions for structured document reuse, however substantial work remains toward the goal of achieving a comprehensive solution. We would like now to extend the current tool. The following lists the directions we will pursue.

4.3.1. Robustness issues

The main purpose is to add new capabilities to the existing tool in order to improve the matching accuracy and thus tool performance.

The current information reuse tool essentially focuses on structural matching, however several works remain to be done in both terminological and constraint based matching. Concerning terminological matching, we made the assumption that element names are descriptive and belong to WordNet as a source of lexical information. However, this assumption does not hold true in practical cases. Similar schema elements in different schemas often have names that differ due to the use of abbreviations, acronyms, punctuations, etc. Techniques to solve such issues have to be integrated into the terminological matching module. Extensions could be inspired for example from [Madhavan 01] where authors propose a normalization step that solves such problems. As part of the normalization step, they perform *tokenization* (parsing names into tokens based on punctuation, case, etc.), *expansion* (identifying abbreviations and acronyms) and *elimination* (discarding prepositions, articles, etc.). In each of these steps they use a thesaurus that can have both common language and domain-specific references. Other direction to be exploited is word soundex (an encoding of names based on how they sound rather than how they are spelled) [Bell 01].

Moreover, exploiting synonyms and hypernyms relationships requires the use of thesauri or dictionaries such as WordNet. In addition, terminological matching can also make use of domain- or enterprise- specific dictionaries containing common names, synonyms, abbreviations, etc. These specific dictionaries require a substantial effort to be built up in a consistent way. However, the latter effort is well worth the investment, especially for schemas with a relatively flat structure where dictionaries provide the most valuable matching hints [Rahm 01]. In this context we plan to use CoP-independent ontologies making explicit CoP's knowledge. Resources useful for CoPs could be annotated semantically with respect to these ontologies. Such annotations could be used to both understand the meaning of elements and make explicit semantic relationships between elements within structured document. Such explicit knowledge could then be used in order to improve the accuracy of terminological matching.

In constraint based matching, we limit ourselves to the analysis of datatype compatibility. However, several constraints such as uniqueness and integrity constraints could give additional hints about matching candidates. Techniques such as the ones described in [Miller 01] and [Li 00] could be used. Moreover, our datatypes constraints analysis is limited to some facets. For example, we do not consider patterns comparison. Research in regular expressions and pattern matching could be a good candidate to extend our work.

Another direction of extensions concerns the kind of documents that we process. We can distinguish between two types of documents with different requirements: *document-centric* and *data-centric* documents. Data-centric documents are documents that use XML for data transport. Although XML is human readable, data-centric documents are designed for machine consumption. Data-centric documents are characterized by a fairly regular structure, fine-grained data, and mostly no mixed content. The order in which elements occur is generally not significant. Document-centric documents are documents that are designed for a human reader. They are characterized by a less regular structure, coarse grained data, and lots of mixed contents. The order in which elements occur is almost always significant, particularly when the document is read serially by a human being.

The current tool essentially focuses on data-centric documents. We plan to extend it to deal with document-centric documents. Issues like elements order and elements with mixed content will be of major interest.

One of the remaining issues is the task of evaluating the performance of the reuse information tool. We proposed an empirical evaluation of the current tool. We used a data set from real world application (bibliographic data). For both direct and complex matches, our solution yields about 95% for both recall and precision. We further compared our work with Cupid and Similarity Flooding algorithms and showed that it outperforms these algorithms [Boukottaya 04]. In the future, we would like to evaluate our schema matching solution using a broad set of applications and data. It will be also important to quantify the reduction in user involvement that a matching solution achieves. Moreover, we plan to run further experimental studies to show the impact of our chosen parameters on the matching process. Finally, the matching process is known to be a heavy and costly process (especially in term of runtime) which is not acceptable for communities like CoPs.

Reduction of runtime will be very important. This could be done for example by allowing CoPs to choose a set of matching criteria that are important for them which avoid to run the whole process.

4.3.2. Development of efficient user interface

Experience suggests that fully automated schema matching is infeasible, especially for complex matches that involve transformation operations. Schema matching solutions must interact with the user in order to arrive at correct final mappings. One of the most important open problems is then efficient user interaction. Specific user input could be for example interactively requested at critical points where it is maximally useful, not just at pre- and/or post-match. This makes post-match editing much easier, since bad guesses made without user guidance need not be corrected and do not propagate. Moreover, the great growing of web data sharing systems will further exacerbate efficient user interaction problems. In fact, even if a near perfect matching solution exists, the user still has to validate the huge number of mapping results produced. The key is then to discover how to minimize user interactions but maximizing the impact of the feedback. A key challenge is to design and implement a simple (an end-user not familiar with schema languages and XML technologies should be able to use such interfaces) and efficient (points where the user interacts with the tool should be clear) user interface. Moreover, a detailed documentation (easy to read and understand) should be provided to end users.

4.3.3. Web services matching

Web services are defined as loosely coupled, reusable software components that refer to programmatic interfaces used in the web for application-to-application communication. The paradigm of web services has been changing the web from a repository of data into a repository of services. This promises universal interoperability and integration since web services enable platform independent and language neutral interaction on the web, as they rely on open XML-based standards like SOAP and WSDL. A critical step in the process of reusing web services for building web-based applications is the discovery of potentially relevant services. UDDI is an industry effort to provide directory services for web services. It provides a standardized set of categories for organizing web services. The current UDDI repositories only enable users to search services based on keyword specifications. This method is clearly insufficient in the context of the PALETTE project first because it requires a shared understanding of the application domain between the service provider and the service consumer. Second, if users are not familiar with the pre-defined service categories, they usually cannot retrieve the required services. Moreover, keywords do not suffice for accurately specifying users' information needs. How to conveniently, accurately and efficiently discover required web services from expanding service repositories will be a critical issue in our work.

We will essentially focus on proposing a set of complementary methods which can be used to support a more automated web services reuse. The intuition underlying our future work is that a plausible means of discovering web services to be reused is to provide a (potentially partial) specification of the desired service (so-called *virtual web service*). Specification of virtual web service includes various aspects, such as textual description, datatypes, and input/output parameters of desired operations. We will propose a set of extensions to matching techniques used within the current information reuse for computing similarities between two WSDL specifications. Similarity computations serve for identifying potentially useful services and estimating their relevance to consumer needs (specified in a virtual web service).

The first extension consists on adapting the current matching techniques by adding matching criteria dealing with web services syntactic, structural and semantic specificity. The comparison of two web services specification is a multi-step process involving the comparison of operation sets offered by the services, which is based on the comparison of operations' input and output messages, which, in turn is based on the comparison of the data types of the objects communicated by these messages. Data types are described using XML schema, thus the current matching techniques will be used at this level. Once relevant services are located, users should be able to integrate with those services automatically. For this, we also aim to extend our XSLT generator to deal with web services transformations.

The second main extension deals with presentation issues: relevant web services should be presented to the end-user in a manner that helps him to choose among several candidates. For this purpose, we plan to derive a set of semantic links between web services in order to structurally organize them into an overlay network (so-called web services repository). Web services will be linked based on their semantic similarities. Compared with current approaches, the advantages of the proposed approach include two aspects: (1) *Convenience:* Web service networks could be represented visually which helps users to focus their intention and flexibly describe their retrieval destination following semantic links. (2) *Efficiency:* Efficiency is obtained by limiting the search space to relevant services (those having a semantic link).

5. Conclusion

Two complementary approaches to document models are presented in this report, as well as a document reuse tool.

Both approaches to document models are intended to make authoring tools easier to use, while producing high quality documents in standard, XML-based formats. Both aim at making format-oriented tools more user friendly. The existing versions of the editing tools involved in this task, Amaya and LimSee2, are format-oriented in the sense that the main goal in their design was to implement document formats. Special attention was paid to the validity of the generated code and to the availability of all features offered by the formats. To some extent, this was done to the detriment of the user, as the original user interface was designed following closely the specification of the document formats.

The user-oriented approach that is taken in this report aims at addressing this issue, without giving up the format-oriented approach and its advantages. It consists in presenting the user with a document model that is closer to him/her, which contains the elements he/she has in mind when producing a document, not (only) the elements of the output format. The user interacts with the tool in terms of these elements and does not bother with the details of the output format. The role of the authoring tool is to transform actions performed by the user in terms of the authoring model into actions on the output format.

This principle is applied to Amaya and LimSee in two different ways, because the kind of documents they handle is very different. Amaya handles only discrete media, whose logical structure has to be mapped only on the display space. LimSee handles continuous media and has to cope with both the display and the synchronization of documents. It works on both space and time. Adding the time dimension is a strong change, which requires an additional step for modelling documents.

In Amaya the authoring model is expressed through a template language called XTiger. This language allows authors to define structures that represent exactly the type of document they manipulate. Simultaneously they specify how these structures are translated into the output format.

In LimSee3, things are more complicated. The output format, SMIL, handles two structures simultaneously: the display structure described by regions, and the time structure, described by a hierarchy of temporal operators. This leads to the introduction of an intermediate format, based on semantically rich objects that can represent the logical structure of multimedia documents. Each object includes both spatial and temporal dimensions of the elements it represents. Building on this intermediate level of abstraction, a template model is used to define the various types of multimedia documents authors may manipulate. This template model plays the same role as the template model in Amaya, but it does not use the output format directly. It relies on the intermediate object model.

In this architecture document, the emphasis is put on the models and the languages to express the models. Only some indications are given about the implementation and the user interface of the editing tools. This will be done in the next step: the document models and languages will now be implemented as an extension of Amaya and as a new version of LimSee.

While the authoring tools are intended to create and update documents, the information reuse tool helps to use existing documents. It improves document usability by providing an automatic transformation process. More precisely, it compares the schema of an existing XML document with another schema describing the target structure of a transformation. As a result of this comparison the structures of the source schema are mapped onto the structures of the target schema and an XSLT transformation script is generated from the mapping. A prototype tool is available.

The last section of this report describes this prototype and its principle of operation, as well as the extensions that are required to make it an operational tool. Three directions of extensions are presented: robustness, user interface and web services. Robustness will be achieved by improving the matching accuracy. Several extensions to mapping techniques are analyzed. Terminological comparisons and constraint matching will be extended. The specific aspects of document-centric structures will be considered and performance will be improved by a careful evaluation of matching methods. The user interface work will focus on user input that may help the matching process to make critical decisions and user contribution to the validation of the generated transformations. Finally, extensions for web services will address matching techniques suited to the specific transformations needed for discovering web services and techniques to help users to choose from the available services.

6. References

[Altheim 01] M. Altheim, S. McCarron, XHTML 1.1 - Module-based XHTML, W3C Recommendation, 31 May 2001, http://www.w3.org/TR/xhtml11/

[Boukottaya 04] A. Boukottaya, Schema matching for structured document transformations, PhD thesis, EPFL, Oct. 2004

[Boukottaya 05] A. Boukottaya, C. Vanoirbeek, Schema matching for transforming structured documents, Proc. 2005 ACM Symposium on Document Engieering, ACM Press, Nov. 2005, pp. 101-110

[Bell 01] G.S. Bell, A. Sethi, Matching records in a national medical patient index, CACM 44(9), 2001, pp. 83-88

[Bray 04a] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 4 Feb. 2004, http://www.w3.org/TR/REC-xml/

[Bray 04b] T. Bray, D. Hollander, A. Layman, R. Tobin, Namespaces in XML 1.1, W3C Recommendation, 4 Feb. 2004, http://www.w3.org/TR/xml-names11/

[Bulterman 05] D. Bulterman et al. Synchronized Multimedia Integration Language (SMIL 2.1), W3C Recommendation, 13 Dec. 2005, http://www.w3.org/TR/SMIL/

[Carlisle 03] D. Carlisle, P. Ion, R. Miner, N. Poppelier, Mathematical Markup Language (MathML) Version 2.0 (Second Edition), W3C Recommendation, 21 Oct. 2003, http://www.w3.org/TR/MathML/

[Clark 99] J. Clark, XSL Transformations (XSLT) Version 1.0, W3C Recommendation, 16 Nov. 1999, http://www.w3.org/TR/xslt

[Fallside 04] D. Fallside, P. Walmsley, XML Schema Part 0: Primer, Second Edition, W3C Recommendation, 28 Oct. 2004, http://www.w3.org/TR/xmlschema-0/

[Ferraiolo 03] J. Ferraiolo, J. Fujisawa, D. Jackson, Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation, 14 Jan. 2003, http://www.w3.org/TR/SVG/

[Hirst 98] G. Hirst, D. St-Onge, Lexical chains as representations of context for the detection and correction of malapropisms, in Christiane Fellbaum (editor), WordNet: An electronic lexical database, Cambridge, MA, The MIT Press, 1998

[Khare 06] R. Khare, Microformats: the next (small) thing on the semantic web?, Internet Computing, IEEE, vol. 10, issue 1, Jan.-Feb. 2006, pp. 68- 75

[Leinonen 03] P. Leinonen, Automating XML Document Structure Transformations. In Proc. ACM Symposium on Document Engineering, ACM Press, 2003, pp. 26-28

[Li 00] W. Li, C. Clifton, SEMINT: A tool for identifying attribute correspondence in heterogeneous databases using neural networks, Data and Knowledge Engineering, 33, 2000, pp. 49–84

[Lie 05] H. W. Lie, B. Bos, Cascading Style Sheets, Designing for the Web, 3rd edition, Addison Wesley, 2005

[Madhavan 01] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with Cupid. Proc. International Conference on Very Large Databases (VLDB), 2001

[Melnik 02] S. Melnik, H. Garcia-Molina, E. Rahm. Similarity Flooding: A versatile Graph Matching Algorithm and its Application to Schema Matching, Proc. 18th International Conference on Data Engineering, 2002

[Miller 01] R. Miller, The Clio Project: managing heterogeneity, ACM SIGMOD Record 30(1), 2001, pp. 78-83

[Quint 04] V. Quint, I. Vatton, Techniques for Authoring Complex XML Documents, Proc. 2004 ACM Symposium on Document Engineering, Oct. 2004, pp. 115-123

[Quint 05] V. Quint, I. Vatton, Towards Active Web Clients, Proc. 2005 ACM Symposium on Document Engieering, ACM Press, Nov. 2005, pp. 168-176

[Rahm 01] E. Rahm, P.A. Bernstein, On matching schema automatically, Microsoft Research Publications, 2001, http://www.research.microsoft.com/pubs

[Su 01] H. Su, H. Kuno, E.A. Rundensteiner, Automating the transformation of XML Documents, Proc. ACM Symposium on Document Engineering, 2001