Malan: A Mapping Language for the Data Manipulation

Arnaud Blouin
GRI - ESEO
Angers, France
arnaud.blouin@eseo.fr

Olivier Beaudoux
GRI - ESEO
Angers, France
olivier.beaudoux@eseo.fr

Stéphane Loiseau
LERIA, University of Angers
Angers, France
stephane.loiseau@univ-angers.fr

ABSTRACT

Malan is a MApping LANguage that allows the generation of transformation programs by specifying a schema mapping between a source and target data schema. By working at the schema level, *Malan* remains independent of any transformation process; it also naturally guarantees the correctness of the transformation target relative to its schema. Moreover, by expressing schemas as UML class diagrams, Malan schema mappings can be written on top of UML modellers. This paper describes the overall approach by focusing on the Malan language itself, and its use within a transformation process.

Categories and Subject Descriptors

I.7 [**Document and text Processing**]: Document Preparation—*Markup languages*; E.0 [**Data**]: General

General Terms

Design, languages

Keywords

mapping, Malan, UML, data manipulation, schema transformation, schema translation

1. INTRODUCTION

The *data manipulation* domain has greatly evolved during the last decade in order to answer new needs. On the first hand, the diversification of communication devices (*e.g.* mobile phone and PDA) requires efficient transformation techniques to display Web pages in an appropriate way that suits the device used. On the other hand, transformations are useful to create presentations from a data set (database and XML document), or to manage the interoperability between documents and databases.

In a standardisation effort, XML [29] has grown to be the standard for storing and organising data. Along with this evolution, a non-negligible number of languages have appeared to manipulate XML documents. Amongst them, XSLT [31] is certainly the most well-known. XSLT manipulates XML documents in order to extract data, to transform documents, or to create presentations. However, the direct manipulation of data, *i.e.* schema instances, is error-prone. Firstly, there exists a dependency on the transformation process; if we wish to use a different transformation language, we have to rewrite the transformation program. Secondly, it does not guarantee the correctness of the transformation process. Working at the schema level by specifying schema mappings avoids such drawbacks. In our previous work [8], we started the application of the mapping concept to document transformation. This paper presents the continuation of this work by presenting our final framework dedicated to the data manipulation domain.

Schema mapping is a concept that allows the definition of relations between two schemas, bringing interoperability to these schemas and consequently to applications that use them [16]. In this paper, we call *mapping* a correspondence that composes a schema mapping: a schema mapping is a set of mappings. A *schema* is a structure that represents a design artefact, such as a relational schema, a UML model, an XML-schema or a DTD [7]. A schema is also called a model in the MDA domain [18]. Since it concerns the manipulation of schemas instances, a transformation program can be considered as an instance of a schema mapping, as illustrated in figure 3. The schema mapping concept is already used in the database domain to facilitate the integration and management of databases [7, 24].

In this paper, we present a framework for manipulating data by *specifying* a schema mapping using our mapping language **Malan** (*a MApping LANguage*). The main idea is to separate the transformation process from the mapping process. *Malan* is a declarative and imperative language that allows the definition of a schema mapping between two schemas, and more precisely between two UML class diagrams [21]. Once established, a schema mapping can be instantiated in order to get a transformation program, such as an XSLT stylesheet or an eXAcT program [5]. Thus, the establishment of a schema mapping allows the independence of any transformation language. The generated transformation program can be applied on a source data set, instance of the source schema, to create a target data set, instance of the target schema. This technique avoids the direct manipulation of schema instances, such as XML documents or

DocEng'08, September 16–19, 2008, São Paulo, Brazil.

database tables. Our prototype, written in Java, is freely available under the terms of the GPL licence[1].

This paper is structured as follows: the next section introduces the application domains of the data manipulation problem. Section 3 describes our framework. Section 4 is devoted to the presentation of *Malan*. Section 5 presents an evaluation of Malan. Section 6 outlines related work within the domains concerned with some comments about our framework.

2. APPLICATION DOMAINS

In the context of the Web 2.0, we define data as being either an XML document or a database. Thus, data manipulation concerns the communication between databases, the interoperability between data, and the creation of presentations from databases or documents. These needs can be grouped into two domains, as depicted in figure 1: schema translation and schema transformation.

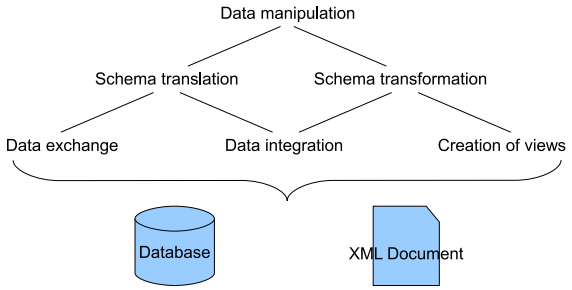


Figure 1: The different application domains of the data manipulation

A data manipulation is referred to as **schema translation** when the semantics of the source schemas is the same as the semantics of the target schema. Schema translation allows the interoperability between heterogeneous data that express a similar concept but in the different formalism (UML, relational schema, DTD, XML schema), or in a same formalism but in a different way. The goal of schema translation is thus to homogenise a set of heterogeneous data. It can be divided into two parts: data exchange and data integration. **Data exchange** [10, 2, 3, 14] consists in translating semi-structured or structured data from a source to a target schema. Figure 2(a) presents an example of data exchange where a *bridge* between an ODF document (*Open Document Format*) and an OOXML document (*Office Open XML*) is created; ODF and OOXML define the same concept of office document, but using two different formalisms.

Data integration [4, 15] consists in combining data from different sources in order to allow a user to get a global and unified view of those data. Figure 2(b) illustrates the principle of the data integration with an example: source databases contain original data while the target regroups the, or a part of the, source data.

Despite that at the origin schema translation mainly concerned the database domain, it was extended to the document domain, as illustrated in figure 2(a). Schema translation is developping since the advent of XML and of the Web 2.0 where a lot of formats, expressed in XML, appeared; such as ODF of OASIS, and Microsoft's OOXML for office documents, or RSS and Atom for Web feed formats.

A data manipulation is a **schema transformation** when the semantics of the source schemas is different than the semantics of the target schema. Schema transformation can be divided into two parts: data integration, and the creation of views. Data integration, presented above as schema translation, can also be considered as schema transformation in some cases. Indeed, if source and target schemas have the same semantics, then it is a schema translation. For example, the fusion of two schemas that define the concept of person, into a global schema that has the same goal, is a schema translation problem. However, given a schema that defines the concept of person, and a schema that specifies the concept of Web navigation; the fusion of these two schemas into a schema that defines a Web navigation statistical model, is considered as a schema transformation problem. In some cases, the classification of a schema integration problem can be subtle and may depend on the context and the semantics of the concerned schemas. The other subdomain of schema transformation is the *creation of views*. In [1], Abiteboul describes views as tools that allow a user to see data from different points of view. A creation of views may have to deal with data integration if different sources have to be used. To illustrate this sub-domain, we can take the example of Web blogs which are composed of a database, containing the posted messages and their comments, and of a presentation created, in most of the cases, as a Web page.

3. FRAMEWORK

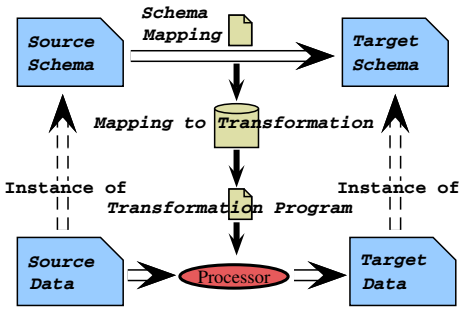
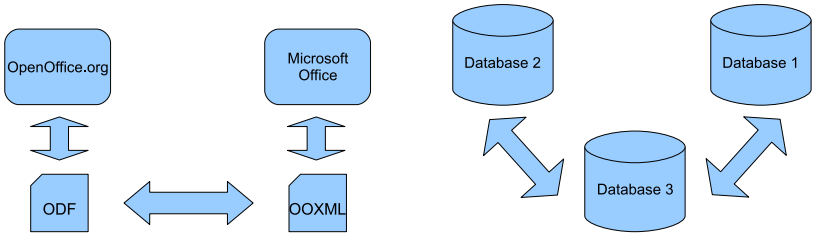


Figure 3: Mapping and Transformation Processes

The global process of our framework is divided into three parts, as illustrated in figure 3. The first part defines the *schema mapping* between a source and a target schema expressed as two UML class diagrams. The goal of the second part is to instantiate the schema mapping previously specified in order to get a *transformation* program. Currently, XSLT stylesheets can be generated but more transformation languages are expected to be managed, such as XQuery [30] or eXAcT [5]. The third part consists in the application of the transformation program to a source schema instance (an XML document for instance), using a transformation process in order to create a target schema instance which will conform to its schema.

Using UML as a schema representation is motivated by the fact that UML is a widely used modelling language for the analysis and design of Information Systems. Because of its popularity, an important number of development and conception tools support it, such as *Netbeans* or *Eclipse*. Moreover, research has been carried out on how to convert

[1]http://gri.eseo.fr/software/malan



(a) Data exchange (b) Data integration

Figure 2: Examples of the data exchange and data integration principles

a UML model to an XML schema and *vice versa* [9, 6], thus facilitating the use of other kinds of schema with Malan. In the same way, UML is also well-suited for databases design [27]. Finally, UML can be easily extended with the use of UML profiles: a profile can define new concepts, called stereotypes, specific to a given domain; for example, the mapping concept, which is not defined in the UML specification, is defined by *Malan* through a UML profile as described in figure 4.

There are two ways to define a *Malan* schema mapping:

- a written *Malan* schema mapping (*e.g.* the schema mappings described in section 5) are given to the *Malan* processor with the two UML class diagrams concerned;
- mappings can be defined by using the *Eclipse* platform[2] with the Eclipse UML plug-in *Papyrus*[3]; with these tools, a set of mappings can be graphically defined between two UML class diagrams. We have created a UML profile, shown in figure 4, that contains a stereotype defining the concept of mapping for UML.

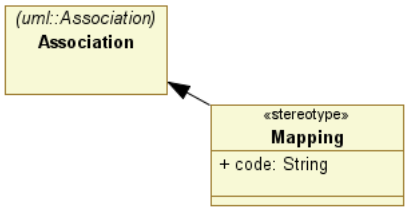


Figure 4: The UML profile for the mapping concept

A mapping is a UML association that has the stereotype <<*mapping*>>. This stereotype has an attribute `code` that contains the *Malan* mapping instructions. Once established, the UML file can be given to the *Malan* processor to generate a transformation.

4. MALAN: A MAPPING LANGUAGE

Malan is a language both declarative and imperative whose structure is divided into three parts:

1. the *schema mapping definition*: this part sets the source and target UML class diagrams concerned by the schema mapping;
2. the *mapping definition*: this part defines mappings between source and target classes of the concerned diagrams, that compose a schema mapping. Each mapping defines mapping instructions between class elements. By class element, we mean either a class attribute, or a class relation named by its role. A mapping is declarative;
3. the *function definition*: as complement to mappings, functions allow to carry out the computation that a mapping may need. A function contains imperative instructions and returns a result.

Figure 5, which defines a schema mapping from a schema that describes a drawing to a schema that describes a boundary, will be used as example in the following sections to illustrate the different features.

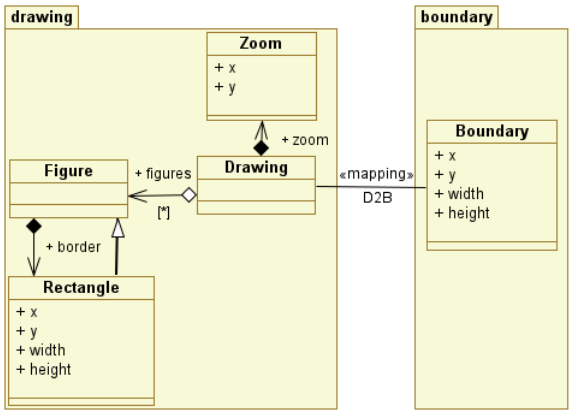


Figure 5: The example *Boundary*

4.1 Schema Mapping Definition

A *Malan* schema mapping is defined by a header and a body; the header defines the URL of the UML class diagrams and the optional name of the schema mapping. The body contains the definition of the mappings and functions. A *Malan* schema mapping must respect the following grammar, using the Backus-Naur Form:

```
(ID ":")? URL "->" URL "{" mappings functions "}"
```

where `mappings` is a set of mappings, `functions` a set of functions, `ID` the name of the schema mapping and `URL` a URL followed by the package name of the class diagram. For example, a possible declaration of figure 5 could be:

[2]www.eclipse.org
[3]www.papyrusuml.org

```
drawing2boundary : "schemas/d2b.uml/drawing" ->
                   "schemas/d2b.uml/boundary"
{
 // Here mappings and functions will be defined.
}
```

where `drawing2boundary` is the name of the schema mapping, `"schemas/d2b.uml/drawing"` and `"schemas/d2b.uml/boundary"` the path of, respectively, the source and the target schema followed by the name of the class diagram package.

4.2 Mapping Definition

As for schema mapping, a mapping is defined by a header and a body; the header defines the optional name and the source and target classes, while the body contains a set of mapping instructions between the classes elements. A mapping must respect the following grammar:

```
(ID ":")? ID ("," ID)* "->" ID "{" instructions "}"
```

where `instructions` is a set of instructions, the first `ID` is name of the mapping, the second and the third `ID` the source classes, and the last `ID` the target class. For example, the `D2B` mapping declaration of figure 5 could be as follows:

```
D2B: Drawing -> Boundary
{
 // Here the instructions will be defined.
}
```

where `D2B` is the name of the mapping, `Drawing` the name of the source class and `Boundary` the name of the target class.

A mapping contains mapping instructions, where each mapping instruction defines a relation between two selected elements *via* the operator "`->`" and must respect the following format:

```
selectedElements "->" selectedElements
```

where the token `selectedElements` is explained in the following paragraphs.

Navigation

From a class, it is possible to access the elements of the other classes, in order to facilitate the mapping definition. The goal of the following code is to get the boundary of a set of figures.

```
1: D2B : Drawing -> Boundary {
2:  min(Drawing.figures.border.x)*zoom.x -> x
3:  min(figures.border.y)*zoom.y -> y
4:  max(figures.border.(x+width))*zoom.x -> width
5:  max(figures.border.(y+height))*zoom.y -> height
6: }
```

Accessing a class attribute or relation can be carried out either with the class name as a prefix (*e.g.* `Drawing.figures.border.x` line 2) or directly (`zoom.x` line 2). The composition `border` defines that a figure has as a rectangular border. Thanks to the navigation, it is possible to access `border` and its attributes from the class `Figure`; for instance, `Figure.border.x` accesses the attribute `x` of the border of a figure. `Figure.border.x` returns only one value since the cardinality of `border` is 1. For a cardinality greater than 1, the returned value is the set of elements that concern the relation; for example, `Drawing.figures.border` returns the list of borders for all the figures concerned by the aggregation `figures`. Line 6 contains the instruction `Drawing.figures.border.(x+width)`; it allows to get the list of the sums `x+width` from the border of each figure, `x` and `width` being `border` attributes. The goal of this navigation feature is to carry out computation while selecting elements.

It is also possible to navigate into relations, since a relation can be considered as a list. Such a navigation is described by the following format:
`ID "[" expression "]"`. Where `ID` is the name of the list concerned and where `expression` must return a value between 1 and `|ID|` inclusive. The syntax `|ID|` corresponds to the cardinality of `ID`. It is also possible to select several elements by using a selection interval which must be put at the end of an instruction, respecting the following format: `"," ID "=" expression ".." expression`, where both `expression` tokens define the interval of the variable `ID`. For example using figure 5:

- `figures[1]`, selects the first figure in the aggregation `figures`;
- `figures[|figures|]`, selects the last figure in `figures`;
- `figures[i], i=1..|figures|`, selects all the figures in `figures`;
- `figures[i], i=|figures|..1`, selects all the figures in `figures` but in the reverse order;
- `figures[j], j=1..|figures|/2`, selects the first half of figures in `figures`;
- if the ordering has no importance, we can directly write `figures`, which is equivalent to `figures[i], i=1..|figures|`.

Mapping of relations

Some special instructions are defined to allow the definition of a mapping that use relations. Figure 6 presents an anonymous example of a mapping of relations where the mapping `A2C` defines the relation between the associations `bs` and `ds` of respectively the classes `A` and `D`. The mapping `B2D` defines the relation between the classes `B` and `D`.

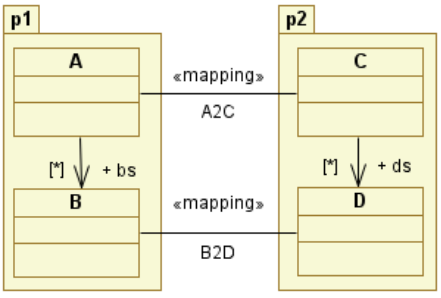


Figure 6: An example of a mapping of relations

The mapping of a relation is divided in two parts: its cardinality definition and the optional ordering of the target

relation elements. The previous paragraph concerning navigation explained that it is possible to select some elements from a list, this principle is used to order target relation elements as illustrated in the different following definitions of A2C:

```
1: A2C : A -> C {
2: 1 -> |ds|
3: bs[1] -> ds[1]
4: }
```

Line 3 of the previous code creates a mapping between the first element of both lists. The cardinality of ds is set to 1 at line 2. The next example presents another possible definition of the mapping A2C.

```
1: A2C : A -> C {
2: bs -> ds
3: }
```

Line 2 of the above code is equivalent to bs[i] -> ds[i], i=1..|bs|. It establishes a mapping from each element of bs at the position i to each element of ds at the same position, with $i \in [1, |bs|]$. The following example describes a last possible definition of the mapping A2C.

```
1: A2C : A -> C {
2: |bs|/2 -> |ds|
3: bs[i*2-1] -> ds[i], i=1..|ds|
4: }
```

This code puts in relation each element of bs being at an odd position ($i*2-1$) with the element of ds at the position i, with $i \in [1, |ds|]$. Line 2 defines that the cardinality of ds is the half of the cardinality of bs.

4.3 Functions

As complement to mappings, functions allow to carry out the computation that a mapping may need. Firstly, Malan provides basic and useful predefined functions to users, such as max, min, or abs for arithmetic calculation; sort, invert, sub for set manipulation; or concat, toLowerCase, length for string manipulation. The definition of the semantics of these predefined functions is out of the scope of this paper.

Secondly, a user can define her own functions which can be called by another functions, or by mappings. Functions are totally imperative, and their syntax is very close to Java or C#. The grammar of a function's header is defined as follows:

```
1: "function" TYPE ID
2: "(" TYPE ID ("," TYPE ID)* ")"
3: "{" FCT_INS "}"
```

where at line 1, TYPE defines the type of the returned value, and ID the name of the function; line 2 defines the parameters where TYPE ID defines respectively the type and the name of a parameter. FCT_INS corresponds to the body of the function; its grammar is too complex to be defined in this paper.

For example, the predefined function sub, that creates a list with elements of a given list from a start to an end position, is defined as follows:

```
1: function list sub(list l, int start, int end){
2:   if(start>end || start<1 || end>|l|)
3:     error(``Invalid parameter(s)'');
4:
5:   list sub = nil;
6:
7:   for(int i=start; i<=end; i++)
8:     sub = sub + l[i];
9:
10:  return sub;
11: }
```

Line 1 of the above code defines the header of the function sub, that returns a non-typed list. It takes a non-typed list, the start and the end position of the elements to take, as parameters. Line 2 checks the parameters, and if the test fails, line 3 raises on error and stops the process. Line 5, the token nil means that the list sub is empty. This list is filled with the wanted elements of the source list in the for loop, lines 7 and 8. The operator + used in line 8 means that element l[i] is added at the end of the list sub. Line 10 returns the final sub-list.

During the instantiation of a schema mapping as a transformation program, the functions used by the schema mapping are also instantiated, when possible, in the target transformation language. For a predefined function, if there exists a semantically equivalent function in the target language, then this equivalent function is used; otherwise the function is instantiated.

4.4 Types

Malan provides six types:

- **integer**, **float**, **string**, and **boolean** types are semantically equivalent to those offered by the Java language. Variables of these types can be declared in functions using the keywords int, float, string, and bool;
- a **class name** defined in a UML class diagram can be used as a type; for example in figure 5, we can define in a function a variable f of class Figure as Figure f;
- the **list** type is notably used to represent elements of relations with a $0..n$, or a $1..n$ cardinality. A list content can be typed; for example the type of the aggregation figures of figure 5 is list<Figure>, where <Figure> defines the type of the elements of the list. The grammar of the declaration of a list is:

  ```
  "list" ("<" TYPE ">")? ID
  ```

  where TYPE is the type of the list elements, and ID the name of the list. Operations on lists are semantically defined by the following inference rules:

$$\text{(lists-concat)}\ \frac{\vdash e_1 : list < \tau > \quad \vdash e_2 : list < \tau >}{\vdash e_1 + e_2 : list < \tau >}$$

$$\text{(begin-concat)}\ \frac{\vdash e_1 : \tau \quad \vdash e_2 : list < \tau >}{\vdash e_1 + e_2 : list < \tau >}$$

$$\text{(end-concat)}\ \frac{\vdash e_1 : list < \tau > \quad \vdash e_2 : \tau}{\vdash e_1 + e_2 : list < \tau >}$$

$$\text{(list-sub)}\ \frac{\vdash e_1 : list < \tau > \quad \vdash e_2 : list < \tau >}{\vdash e_1 - e_2 : list < \tau >}$$

Class name κ

Type $\tau ::= int \mid float \mid bool \mid string$
$\mid list$
$\mid list < \tau >$
$\mid list < \kappa >$
$\mid \kappa$

where, `lists-concat` corresponds to the concatenation of two lists of the same type; `list-sub` to the deletion of the elements of e_2 in e_1; `begin-concat` and `end-concat` to the addition of an element of type τ, respectively, at the beginning and at the end of a list of the same type τ. Concerning non-typed lists, their inference rules are equivalent to the above, but $list < \tau >$ is replaced by $list$.

5. EVALUATION

In this section, we evaluate Malan by comparing it to, as far as we know, the main mapping framework found in literature: Clio.

Clio [17, 24, 11] is an IBM Research system for expressing declarative schema mappings using a graphical interface. Figure 7 presents the interface of Clio, in which the mapping component is divided in two elements; the left and the right parts contain respectively, the source and the target schemas. Clio manages relational schema, XML schema, and DTD which are shown in a nested relational representation. As opposed to Malan, where mappings are defined between classes and then between their attributes, Clio specifies mappings only between attributes. Once established, mappings are compiled into a query graph representation, which can be instantiated as a transformation program (XQuery, XSLT, SQL, or SQL/XML).

The three following sections present specific data manipulation examples that compare Malan and Clio. Section 5.4 sums up the ability of these two frameworks to answer to the data manipulation problem.

5.1 Example 1: a schema transformation

The first example consists of a schema transformation; given a source schema that defines a poem, we want to create a presentation of it in the SVG format. Figure 8 illustrates a possible SVG presentation of a poem. Lines 1 and 2, the `svg` tag defines the beginning of the drawing. The `g` tag, at line 3, corresponds to a group of SVG shapes where its attributes are applied on every nested shapes. Lines 4 and 5, defines the rectangle that boxes the poem, followed by the definition of the title of the poem at lines 6 and 7. Lines 8 and 9 define another group of shapes that contains the verses of the poem (from line 10 to line 13).

The poem schema, shown in the left part of figure 7, is composed of a `poem` that has a `name`, and a list of `verses`, where a `verse` is a string. The SVG schema is available on the W3C Website[4].

The main difficulty of this example is the computation of the layout needed to place shapes into the drawing: to define the height of the drawing, we need to know the number of verses. Moreover, to place each verse, we must know its position within the poem. To do such operations, functions that operate on lists and their items are necessary. Clio does not provide such functions, thus illustrating its limits concerning schema transformation. However, to bypass this drawback, we can directly call the necessary XSLT functions using the Clio expression editor. For example, the Y-coordinate of a verse can be defined using an XSLT expression, *i.e.* `"{position()*20+40}"`, as depicted in figure 7. The drawback of this process is that the schema mapping is dependant of the target transformation language, which is in contradiction with one of the goal of the mapping paradigm.

[4] http://www.w3.org/Graphics/SVG/1.2/rng/Tiny-1.2/Tiny-1.2.rng

```
1 <s:svg xmlns:s="http://www.w3.org/2000/svg" width="100%"
2   height="100%">
3  <s:g transform="scale(2) translate(100,50)">
4   <s:rect x="-50" y="0" width="300" height="180" rx="10"
5    fill="rgb(120,0,0)" stroke="rgb(40,0,0)" stroke-width="2"/>
6   <s:text x="-40" y="40" fill="white"
7    font-weight="bold">La flamme</svg:text>
8   <s:g transform="translate(100,20)" font-style="italic"
9    text-anchor="middle" fill="white">
10   <s:text y="70">Vêtue de jaune, vêtue de bleu,</svg:text>
11   <s:text y="90">Se croyant plus forte qu'un feu,</svg:text>
12   <s:text y="110">La belle flamme s'enhardit,</svg:text>
13   <s:text y="130">Sur son destrier la bougie.</svg:text>
14  </s:g>
15 </s:g>
16 </s:svg>
```

Figure 8: An SVG presentation

```
1 schemas/poem.uml -> schemas/svg.uml {
2   poem2svg : Poem -> svg {
3     "100%" -> width
4     "100%" -> height
5     "scale(2) translate(100,50)" -> g.transform
6     -50 -> g.rect.x
7     0   -> g.rect.y
8     300 -> g.rect.width
9     100+|verses|*20 -> g.rect.height
10    "rgb(120,0,0)"  -> g.rect.fill
11    10  -> g.rect.rx
12    "rgb(40,0,0)" -> g.rect.stroke
13    2 -> g.rect.stroke-width
14    -40 -> g.text.x
15    40  -> g.text.y
16    "white" -> g.text.fill
17    "bold"  -> g.text.font-weight
18    name    -> g.text.#textContent
19    "translate(100,20)" -> g.g.transform
20    "italic" -> g.g.font-style
21    "middle" -> g.g.text-anchor
22    "white"  -> g.g.fill
23    verses   -> g.g.text
24  }
25  verse2text : verse -> text {
26    position(verse)*20+50 -> y
27    #textContent -> #textContent
28  }
29 }
```

Figure 9: The *poem2SVG* Malan schema mapping

Figure 9 corresponds to the Malan schema mapping for the *poem to SVG* example. Line 1 defines the UML class diagrams used for the schema mapping. It is composed of two mappings: `poem2svg` line 2, and `verse2text` line 25. `poem2svg` defines a mapping from a poem to an SVG element. From line 6 to line 13, the rectangle that frames the

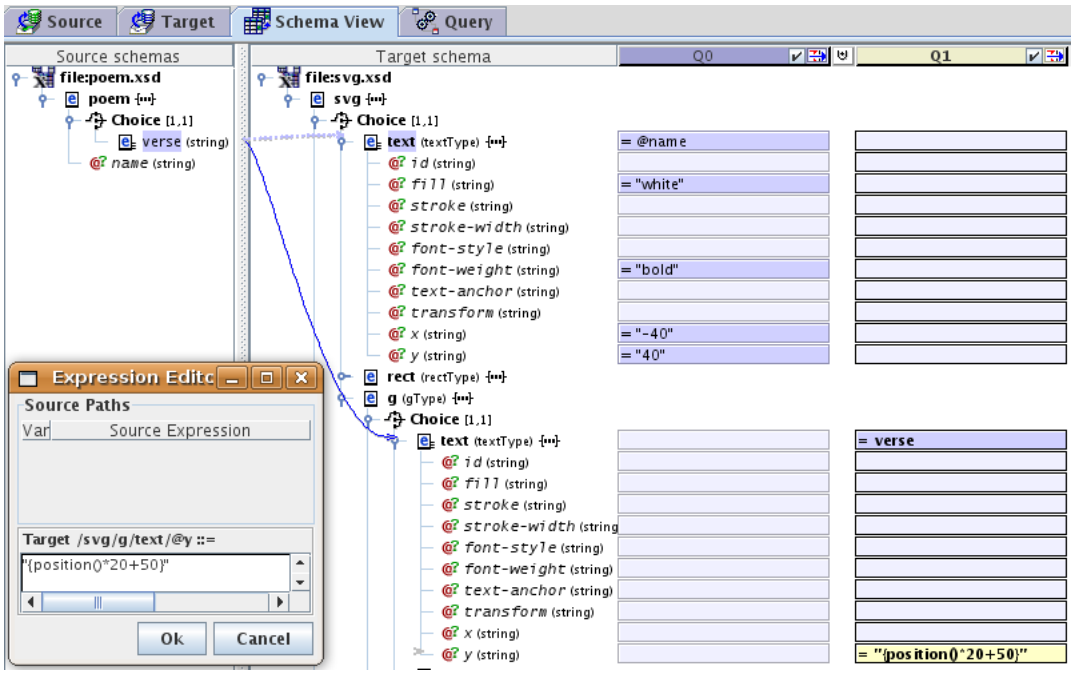


Figure 7: The Clio interface

poem is specified using an SVG `rect` element. The instructions from line 14 to lines 18, that have as target an SVG `text` element, corresponds to the definition of the title of the poem. From line 19 to line 22, the `g` element that contains the verses is defined. Line 23 specifies that every `verse` of the list `verses` corresponds to an SVG `text` element in `g.g`. This line is equivalent to

```
verses[i] -> g.g.text[i], i=1..|verses|
```

as explained in section 4.2; it means that for the verse at position `i`, there must exist a text element at the same position. This process is necessary since the position of a verse in a poem is essential. Concerning complex instructions, line 9 states that the height of the rectangle surrounding the poem, depends of the number of verses (`100+|verses|*20 -> g.rect .height`). Moreover, line 26, the function `position(verses)` returns the position of the current `Verse` in the list `verses`. Thus we can compute the position of each verse into the drawing.

This schema transformation example, which only has a low complexity, shows the limitations of a pure declarative mapping language that does not provide complex programming features, such as function definition or set manipulation.

5.2 Example 2: the Turing machine example

The goal of this second example is to show the expressiveness of Malan by implementing a universal Turing machine. A universal Turing machine is a Turing machine that can model any Turing machine. A Turing machine is composed of:

- a **tape**, that contains cells; each cell contains a symbol of a given alphabet. The *white symbol* is a special symbol used to set the default value of a cell. A tape is indefinitely extendable to the right and to the left;
- a finite set of **states**; the initial state q_0 is a start state, and the process stops when it reaches an end state (see figure 10);
- a **head**, that reads and writes cells symbols, and moves to the right or to the left cell of the current cell;
- a set of **actions**; an action is executed when the current state and symbol, read by the head, match respectively with the entry state and symbol of the action. The execution of an action replaces the current state and symbol by an output state and symbol. Then, depending on the action, the head moves to the right or to the left of the current cell.

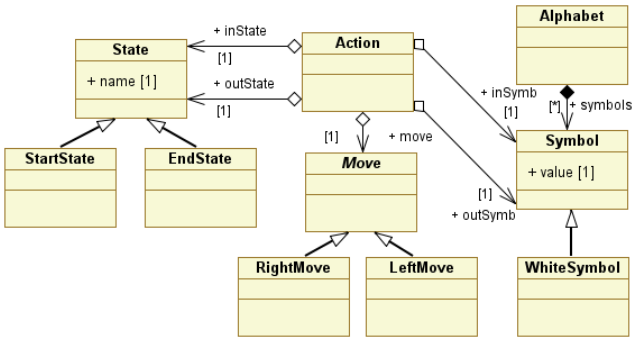


Figure 10: Our Turing machine UML class diagram

Figure 11 is a Malan function that defines a Turing machine. The function takes as input (lines 1 and 2), an initial tape T, that may contain some cells, a start state q_0, and a set of actions A. The function returns the resulting tape which consists of a list of symbols. Lines 3 and 6 declare

the variables used in the function, where `tape` is the tape that will contain the results of the execution of the Turing machine, `currState` the current state, `position` the current position of the head on the tape, and `end` a boolean that states if the process must stop. The machine consists of a *while* loop (from line 8 to line 33), that ends when the current state is an end state (line 8), or if no action exists for the current state and symbol (see line 25). The first step of the loop is to extend the tape to the right or to the left, with a white symbol, if the current position points to a cell that does not exist yet (from line 12 to line 17). Then, we search an action that matches the criteria (from line 19 to line 23); if no matching action exists, the process stops; otherwise, the current symbol and state are replaced by the output symbol and state of the action; then, the head is moved to the right or to the left of the current cell.

```
1 function List<Symbol> turingMachine(List<Symbol> T,
2 StartState q0, Set<Action> A) {
3   int position      = 1;
4   State currState   = q0;
5   bool end          = false;
6   List<Symbol> tape = T;
7
8   while(!end && !(currState is EndState)) {
9     Action a = null;
10    int i    = 1;
11
12    if(position<1) {
13        position = 1;
14        tape     = WhiteSymbol + tape;
15    }
16    else if(position>|tape|)
17            tape = tape + WhiteSymbol;
18
19    while(i<=|A| && a==null)
20        if(A[i].inState.name==currState.name &&
21           tape[position].value==A[i].inSymb.value)
22             a = A[i];
23        else i++;
24
25    if(a==null) end = true;
26    else {
27        tape[position].value = a.outSymb.value;
28        currState = a.outState;
29
30        if(a.move is Right) position++;
31        else position--;
32    }
33  }
34  return tape;
35 }
```

Figure 11: A Turing machine in Malan

This example cannot be defined in Clio since it is a pure declarative mapping language. It shows that the expressiveness of Malan allows to define complex algorithms that mappings may use. We think that such a feature is needed to specify complex schema transformations.

5.3 Example 3: a data exchange problem

This last example presents a data exchange problem used to describe the Clio process in [24]. The two schemas are described in figure 12, where according to Popa *and al.*:

> The left-hand schema represents a source relational schema with three tables: `project(name, year)`, `company(cid, cname, city)`, and `grant (gid, cid, amount, project)`. It describes information about companies, their projects and the grants given for those projects. Each grant is given to a company for a specific project. Therefore, each grant tuple has foreign keys (`cid` and `project`) referencing the associated company and project tuples. The right-hand schema represents a target XML schema. While the information that the target contains is similar to that of the source, the data is structured in a different way. Organisations and projects are grouped by city. For each different city, there is an element `cityStat` containing the organisations and the grants in that city. Project funding data are then nested within `organization` and related with the financial information through a foreign key based on the `aid` element.

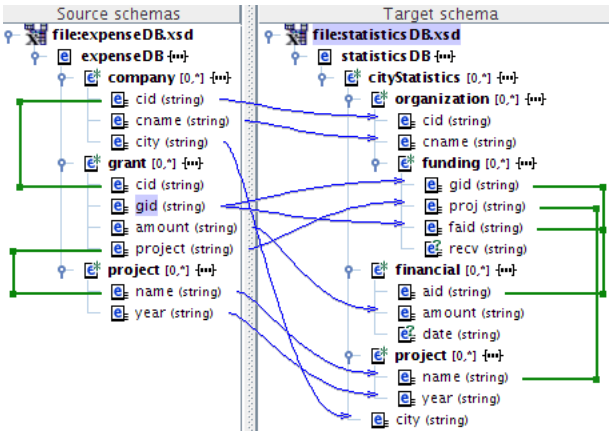


Figure 12: The Clio schema mapping of example 3

Figure 12 presents the schema mapping for this example. It is composed of 9 mappings that were easily established thanks to the similarity between their name. The internal process of Clio directly deduces that organisations must be grouped by city without any help of the user.

```
1 schemas/expenseDB.uml -> schemas/statDB.uml {
2   E2S: expenseDB -> statDB{
3     unique(companies, ‘‘city’’).city-> cityStats.city
4     companies[cityStats[i].city==city] ->
5         cityStats[i].orgs, i=1..|cityStats|
6     grants[project==cityStats.orgs[i].cname] ->
7         cityStats.orgs[i].fundings,
8         i=1..|cityStats.orgs|
9   }
10  G2F: grant -> funding {
11    gid -> gid
12    amount -> faid.amount
13    gid -> faid.aid
14    project -> proj
15  }
16  P2P: project -> project {
17    name -> name
18    year -> year
19  }
20 }
```

Figure 13: The Malan schema mapping of example 3

Figure 13 presents the Malan schema mapping for the current example. It is composed of 3 mappings: the `P2P` mapping defines the relation between the element `project`

of each schema. The mapping `G2F` states that a `grant` corresponds to a `funding` composed of a single `financial` (the attribute `faid` refers to the id of the related `financial`). The most important mapping is `E2S` that puts in relation the two root elements of the schemas. Line 3 defines that the organisations must be grouped by city: function `unique` is a Malan predefined function that, in this case, returns a list containing the cities with no redundancy; each value of the resulting list is put in relation with a `CityStat`. Lines 4 and 5 defines the `companies` that corresponds to the organisations `orgs` of each `CityStat`: for each `CityStat`, we select the companies that have the same city. Lines 6, 7 and 8 defines the `fundings` of each `organization`.

This example shows that concerning data exchange problems, Clio can be more appropriate than Malan: with Malan we have to define the relation of the concerned classes in order to define the relation between their attributes; this process can be complex as illustrated in figure 13. On the contrary with Clio, mappings, defined between attributes, are compiled into an internal representation that catches the semantics of the mappings; thus, it allows, for instance, to deduce that companies must be grouped by city.

5.4 Results

We conclude on this section by the pros and cons of Malan and Clio for the three application domains presented in section 2: data transformation, data exchange, and data integration.

Concerning data transformation, the two first examples show that the expressiveness of Malan allows complex data transformations, on contrary to Clio. We think that to be able to define data transformations, a mapping language has to allow imperative instructions in order to improve its expressiveness; which is limited if only declarative instructions, used to specify mappings, are permitted.

Concerning data exchange; the last example demonstrates that Clio can be more appropriate for this kind of problem. Moreover, one of the main challenge of the data exchange domain is to automatically specify a set of mappings between heterogenous schemas; this problem is called *schema matching*[25]. Currently, the Clio prototype supports semi-automatic schema matching, which simplifies the mapping definition process. Research must be carried out to apply matching techniques on UML class diagrams, in order to implement a schema matching engine in the Malan prototype.

Data integration can only be carry out if the mapping language can manage several source schemas. For the moment, both Malan an Clio prototypes do not support this feature, but it can be easily integrated if we consider the source schemas $S_1 \dots S_n$ as a single schema S_g, as describes in the following formula:

$$\cup_{i=1}^{n} S_i = S_g \quad (1)$$

6. RELATED WORK AND DISCUSSION

To avoid the direct coding of transformation, VXT [23] is a visual programming language specifically designed for programming XML transformations. This point of view is very attractive since it aims at reducing user's cognitive load. However, it has the drawback of being still dependant on the target language since the VXT environment provides a specification mode for each managed target language, *i.e.* XSLT and Circus.

Concerning transformation, ATL [13] (Atlas Transformation Language) and QVT [20] (Query View Transformation) are two transformation languages dedicated to perform transformations within the MDA framework (Model Driven Architecture). Both of them address the model transformation problem by adding a higher abstraction level, the MOF (Meta-Object Facility) meta-meta-model. In their architecture, the source and target models are conformed to their meta-models, and the source and target meta-models are conformed to the MOF meta-meta model. ATL and QVT are dedicated to developers that operates on the MDA domain. Contrary to these two languages, *Malan* is dedicated to the data manipulation, *i.e.* to the database and XML document manipulation; ATL and QVT do not fit to this problem since they address a more general problem than *Malan* which does not suffer of the complexity resulting of MDA.

Representing schema mapping in UML has already been the concern of research from Hausmann *et al.* [12]. They described a UML extension that allows the visualisation of a schema mapping between two diagrams where each mapping is completed by OCL constraints. OCL [19] is not well-suited for the schema mapping definition since it is a language originally dedicated for the constraint definition on UML elements. Its main goal is to enrich a UML class diagram semantically. Even if most of the mapping concept elements can be specified using OCL, it is not as simple as with *Malan* in most cases.

Even if our framework works with UML class diagrams, it is important to keep in mind that in the data domain (XML document and database domains included), schemas are often represented in the XML schema format. Thus, the transition of an XML-schema to a UML class diagram is a non negligible step which provides consequent difficulties and limitations; for instance, UML is aimed at software design rather than data modelling. So to facilitate the transition, a proposed solution could be the definition of a UML profile describing XML-schema properties (attribute, element, complex type) [26].

Moreover, it is important to insist on the fact that *Malan* does not aim to replace the use of any transformation process. In some cases, the use of a mapping process is not appropriate because of its schema-awareness which can be time-consuming. For example, to perform a simple and quick data transformation of which schemas are not available, a user may prefer the direct definition of the transformation program, using XSLT for instance, instead of looking for or defining the schemas to define a schema mapping, even if it is error-prone.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented *Malan*, a mapping language that allows the definition of a schema mapping, that are two UML class diagrams. This schema mapping can be instantiated in order to create a transformation program that allows the transformation of a source schema instance into a target schema instance. This approach aims to facilitate the development process of a transformation by clearly separating the mapping and transformation processes and consequently to be free to use whichever transformation lan-

guage we want. We have illustrated our approach with two examples showing the expressiveness of Malan.

Our future work are twofold and concerns the Web data and the interaction domains. Firstly, we aim to fully apply *Malan* to the data manipulation, *i.e.* to the database and XML document manipulation. The work described in this paper mainly concerns the XML document manipulation. Consequently, we have to extend our framework to the database manipulation. This extension will help us to test *Malan* on more use cases and thus, we will be able to enrich and refine its expressiveness. Secondly, we aim to apply *Malan* to the GUI domain and in particular to the RIA (Rich Internet Application) domain of the Web 2.0. The first step of that work will be the use of *Malan* for the automatic generation of transformations fitted to interactive system, *i.e.* incremental [28, 22] or active [5] transformations. Then, we will be able to define a model that will allow to specify, at the UML level, the interaction, as *Malan* does for the mapping specification.

8. ACKNOWLEDGEMENTS

We thank Lucian Popa for his help on Clio, Richard Woodward and Olivier Camp for their comments on this paper. The work described in this paper has been funded by a grant from *Angers Loire Metropole*.

9. REFERENCES

[1] S. Abiteboul. On views and xml. In *Proc. of PODS '99*, pages 1–9, 1999.

[2] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. *Theor. Comput. Sci*, 275(1-2):179–213, 2002.

[3] P. Atzeni. Schema and data translation. In *Proc. of ICDE '06*, page 103. IEEE Computer Society, 2006.

[4] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[5] O. Beaudoux. XML active transformation (eXAcT): transforming documents within interactive systems. In *Proc. of DocEng '05*, pages 146–148, 2005.

[6] M. Bernauer, G. Kappel, and G. Kramler. Representing XML schema in UML - a comparison of approaches. In *Proc. of ICWE'04*, pages 440–444. Springer-Verlag, 2004.

[7] P. A. Bernstein, A. Y. Halevy, and R. A. Pottinger. A vision for management of complex models. *SIGMOD Rec.*, 29(4):55–63, 2000.

[8] A. Blouin and O. Beaudoux. Mapping paradigm for document transformation. In *Proc. of DocEng '07*, pages 219–221. ACM Press, 2007.

[9] E. Domínguez, J. Lloret, B. Pérez, A. Rodríguez, A. L. Rubio, and M. A. Zapata. A survey of UML models to XML schemas transformations. In *Proc. of WISE'07*, pages 184–195. Springer, 2007.

[10] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2003.

[11] L. M. Haas, M. A. Hernandez, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *Proc. of SIGMOD '05*, pages 805–810, 2005.

[12] J. H. Hausmann and S. Kent. Visualizing model mapping in UML. In *Proc. of SoftVis '03*, pages 169–178. ACM Press, 2003.

[13] F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[14] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proc. of PODS '05*, pages 61–75. ACM Press, 2005.

[15] M. Lenzerini. Data integration: a theoretical perspective. In *Proc. of PODS '02*, pages 233–246. ACM Press, 2002.

[16] J. Madhavan, P. A. Bernstein, P. Domingos, and A. Y. Halevy. Representing and reasoning about mappings between domain models. In *Eighteenth national conference on Artificial intelligence*, pages 80–86, 2002.

[17] R. J. Miller, M. A. Hernandez, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The clio project: Managing heterogeneity. *SIGMOD Rec.*, 30(1):78–83, 2001.

[18] OMG. MDA specification. Technical report, OMG, 2001.

[19] OMG. UML 2.0 OCL Specification. Technical report, OMG, 2003.

[20] OMG. MOF QVT Specification. Technical report, OMG, 2005.

[21] OMG. UML 2.1.1 specification. Technical report, OMG, 2007.

[22] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *Proc. of WWW '05*, pages 671–681. ACM Press, 2005.

[23] E. Pietriga, J.-Y. Vion-Dury, and V. Quint. VXT: a visual approach to XML transformations. In *Proc. of DocEng '01*, pages 1–10, 2001.

[24] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating web data. In *Proc. of VLDB'02*, pages 598–609, 2002.

[25] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[26] N. Routledge, L. Bird, and A. Goodchild. UML and XML Schema. In *Proc. of the Thirteenth Australasian Database Conference*, volume Vol. 5, pages 157–166, January 2002.

[27] C. Soutou. *UML 2 pour les bases de données*. Eyrolles, 2007.

[28] L. Villard and N. Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *Proc. of WWW '02*, pages 474–485, 2002.

[29] W3C. Extensible markup language 1.1 specification. Technical report, W3C, 2006.

[30] W3C. XQuery 1.0: An XML query language. Technical report, W3C, 2006.

[31] W3C. XSL transformations (XSLT) version 2.0 recommendation. Technical report, W3C, 2007.