# Clip: a Visual Language for Explicit Schema Mappings

Alessandro Raffio*, Daniele Braga*, Stefano Ceri*, Paolo Papotti†, Mauricio A. Hernández‡

*Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, Milano, Italy
raffio,braga,ceri at elet.polimi.it

†Dipartimento di Informatica e Automazione, Universitá di Roma Tre
Via della Vasca Navale 79, Roma, Italy
papotti at dia.uniroma3.it

‡IBM Almaden Research Center
San Jose, CA, US
mauricio at almaden.ibm.com

*Abstract*— **Many data integration solutions in the market today include tools for schema mapping, to help users visually relate elements of different schemas. Schema elements are connected with lines, which are interpreted as mappings, i.e. high-level logical expressions capturing the relationship between source and target data-sets; these are compiled into queries and programs that convert source-side data instances into target-side instances. This paper describes Clip, an XML Schema mapping tool distinguished from existing tools in that mappings explicitly specify structural transformations in addition to value couplings. Since Clip maps hierarchical XML schemas, lines appear naturally nested. We describe the transformation semantics associated with our "lines" and how they combine to form mappings that are more expressive than those generated by Clio, a well-known mapping tool. Further, we extend Clio's mapping generation algorithms to generate Clip's mappings.**

## I. Introduction

Schema mappings are created and maintained for two purposes. In the most popular usage scenario, given a schema mapping between a source and a target schema, mapping tools automatically create a transformation script (a query or program) which converts a source side instance into a target side instance. In other words, mapping tools help data integration engineers write long and complex programs to transform data. Another usage of schema mappings is to maintain relationships between schema elements, for later use in impact analysis (change management) and data lineage. However, we do not further consider this problem in the paper.

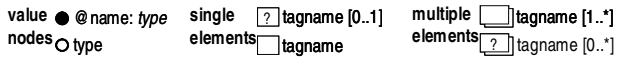
Schema mapping tools are characterized by GUIs that place a source structure on one side of the screen and a target structure on the other side. Users specify the correspondences by drawing lines across the source and target structures. Lines are typically annotated with features that carry some of the transformation semantics (e.g., filtering predicates, functions, etc.). Many such tools are available in the market today (e.g., Altova MapForce, Microsoft BizTalk, Stylus Studio, IBM Rational Data Architect)[1]. Clio [1][2] is a schema mapping prototype developed jointly between IBM Almaden Research Center and the University of Toronto; it automatically creates mappings given two schemas and a set of value mappings between their atomic elements. Clio builds an internal representation capable of representing relational and XML schemas; depending on the kind of source and target schemas, Clio can render queries that convert source data into target data in a number of languages (XQuery, XSLT, SQL/XML, SQL).

This paper introduces *Clip*, a new schema mapping language developed at Politecnico di Milano. Users of Clip enter mappings by drawing lines across schema elements and by annotating some of the lines. Clip then produces the queries that implement the mapping. The main difference introduced by Clip relative to its predecessors, and most specifically to Clio that greatly influenced Clip's design, is the introduction of structural mappings in addition to value mappings; this gives users greater control over the produced transformations. Clip mappings are less dependent on hidden automatisms, whose assumptions may fail in capturing the intended semantics.

Clip was designed to work with XML Schemas. But, just like Clio, Clip also works with relational schemas, as long as they are converted in a canonical way into XML Schemas. In general, Clip should work with any schema model that can be visually represented as a nested containment of relations.

### A. Motivating Example and Approach

XML Schemas are represented as trees; attributes[2] and text are represented by black and white circles respectively, elements by squares, cardinalities by both icons and labels:[3]




[1]On the Web: www.altova.com/MapForce, www.microsoft.com/biztalk, www.stylusstudio.com, www.ibm.com/software/data/integration/rda.

[2]The attribute name, next to the black circle, is preceded by an "@" sign to resemble the XPath symbol for accessing attributes.

[3]Notice that the question marks and the zeroes in the minimum cardinality indicate optionality, while shadowed icons and stars in the maximum cardinality indicate multiplicity.

Referential integrity between attributes is represented by a dashed line (as for @pid attributes of regEmps, which refer to those of Projs).

Consider the following XML document instance describing two departments (compacted due to space limits):

```
source---dept---dname = ICT
     |      |---Proj---@pid = 0001
     |      |      `---pname = Appliances
     |      |---Proj---@pid = 0002
     |      |      `---pname = Robotics
     |      |---regEmp---@pid = 0001
     |      |         |---ename = John Smith
     |      |         `---sal = 10000
     |      |---regEmp---@pid = 0001
     |      |         |---ename = Andrew Clarence
     |      |         `---sal = 12000
     |      |---regEmp---@pid = 0002
     |      |         |---ename = Mark Tane
     |      |         `---sal = 10500
     |      `---regEmp---@pid = 0002
     |                |---ename = Jim Bellish
     |                `---sal = 11000
     `---dept---dname = Marketing
            |---Proj---@pid = 0001
            |      `---pname = Brand promotion
            |---Proj---@pid = 0032
            |      `---pname = Appliances
            |---regEmp---@pid = 0001
            |         |---ename = Richard Dawson
            |         `---sal = 30000
            |---regEmp---@pid = 0032
            |         |---ename = Mark Tane
            |         `---sal = 10000
            `---regEmp---@pid = 0001
                      |---ename = Steven Aiking
                      `---sal = 20000
```

Each department has a name, a list of projects (with an identifier and a name), and a list of regular employees (with a name, a salary, and a @pid attribute referring to the project they work on). This instance is valid w.r.t the source XML Schema on the left of Figure 1. We want to transform the source instance into one of the target schema on the right of Figure 1. The desired output is:

```
target---department---project---@name = Appliances
     |            |---project---@name = Robotics
     |            |---employee---@name = John Smith
     |            |---employee---@name = Andrew Clarence
     |            |---employee---@name = Mark Tane
     |            `---employee---@name = Jim Bellish
     `---department---project---@name = Brand promotion
                  |---project---@name = Appliances
                  |---employee---@name = Richard Dawson
                  |---employee---@name = Mark Tane
                  `---employee---@name = Steven Aiking
```

None of the generation tools we are aware of can obtain such (simple) result. Figure 1 is actually an attempt to express our mapping in Clio. It only gets close to the target, as it compiles to a transformation that outputs projects and employees, but encloses each node in a different department element, not preserving containment and sibling relationships:

```
target---department---project---@name = Appliances
     |---department---project---@name = Robotics
     ...
     `---department---employee---@name = Steven Aiking
```

This happens because structural transformations are inferred from value mappings instead of being under the user's control. The inference rule states, in this case, that a department be generated for each mapped value; however, this is not the desired default behavior. This paper addresses how to express this and more complex mappings, by controlling structural transformations.

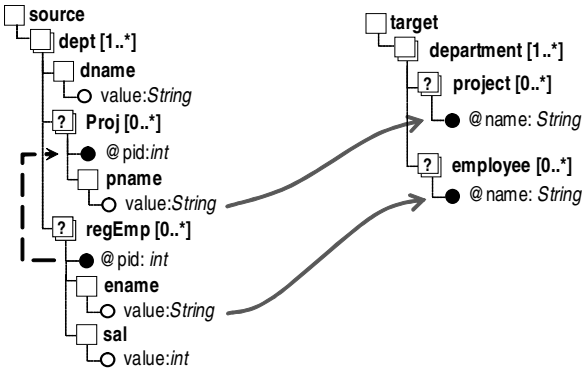



Fig. 1. A simple yet problematic mapping

### B. Related Work

The schema mapping generation problem received lots of attention by the research community in the last years [1][3][4]. Schema mapping generation studies how to automatically create schema mappings given as input two schemas and a set of value mappings between their atomic elements. In essence, mapping generation algorithms ascribe transformation semantics to value mappings generating first order logical formulas that are independent of the actual transformation language and easier to study than executable scripts.

With respect to existing approaches, we allow users to express more complex mappings by introducing a new GUI, extended with set correspondences and aggregate functions. Our contribution tries to bring an initial answer to the growing demand for more complex schema mappings [5]. To generate such mappings we introduce new generation algorithms that generate second-order logical formulas, which allow to express grouping and aggregate by means of functions. Without functions, our mapping language reduces to the language of GLAV mappings [6], or source-to-target tgds (tgd stands for tuple generating dependency) [7]; in the object-relational case, in fact, it is a nested-relational extension of second order tgds [8] and the two languages coincide in the relational case. Nested second order tgds have been already presented in schema evolution scenarios [9] and in the nested mapping paradigm [2]. Strictly referring to the mapping language expressiveness, the main difference with these approaches is that they only consider Skolem functions, while we extend the set of supported functions.

## II. The Clip Language

Clip builds on the visual representation of XML Schemas introduced in Section I-A and uses two different kinds of lines to connect source and target nodes:

*Value mappings* connect value nodes to establish correspondences between atomic values (as in Clio). In general, value mappings convert one or more source values into a target

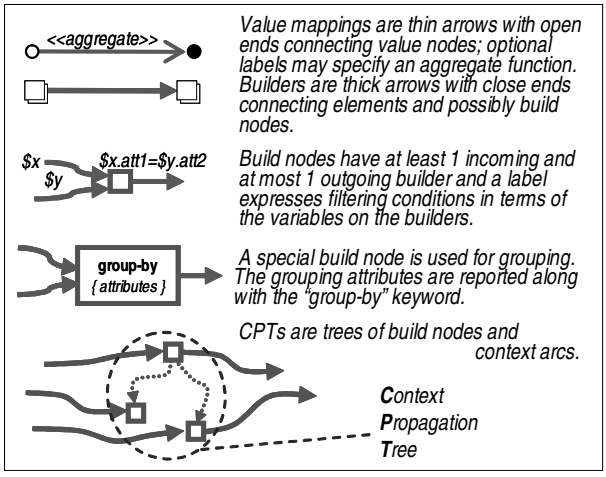



Fig. 2. The Clip syntax in a nutshell

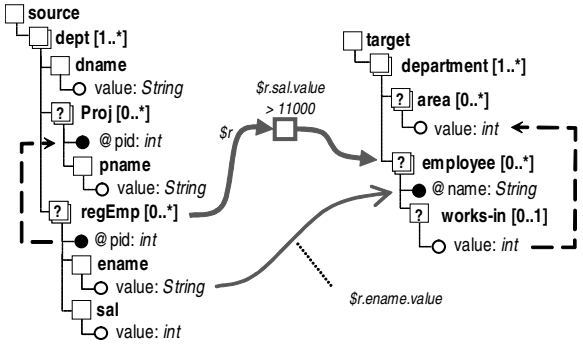



Fig. 3. A simple Clip mapping

value. Simple one-to-one value mappings represent the identity function and copy a single source value as a target value. More complicated transformations require the user to add a scalar function that transform the one or more source values into a target value. For example, value mappings can concatenate multiple source values or perform an arithmetic operation on source values. *Object mappings* (or *builders*) connect elements and rule structural transformations. Intuitively, builders represent iterators on the source nodes they are drawn from: in each iteration, a new element is constructed, of the kind of the target node reached by the builder.

We introduce some further notations. *Build nodes* are annotated nodes placed between the schemas and connected to schema nodes by builders; build nodes can also be connected one to another by *context arcs* into tree structures, named *context propagation trees* (CPTs). Build nodes have 1..n incoming builders, 0..1 incoming context arcs, 0..1 outgoing builders and 0..n outgoing context arcs. Incoming builders may be tagged with *variables*, representative of the nodes from which they originate, if they need to be referenced in node labels. Last, *group* nodes are build nodes marked by a "group-by" label and a list of grouping attributes. Figure 2 summarizes the visual syntax of Clip.

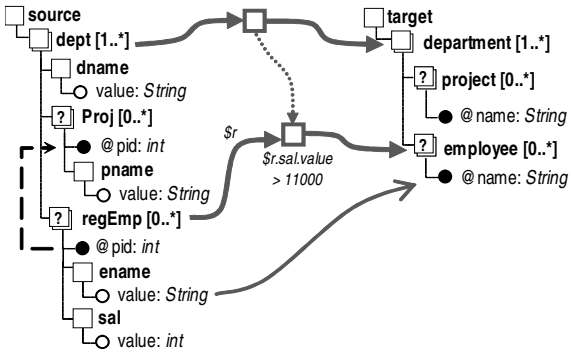



Fig. 4. A mapping with context propagation

### A. Clip Mapping examples

**Simple mapping.** Figure 3 shows a simple mapping, where an employee is created for each regEmp whose salary ($r.sal.value) is greater than 11000. For each such employee, the name (ename) is copied to the name attribute (a value mapping connects ename.value to @name). Note the condition expressed using variable $r taken from the incoming builder. This mapping is indeed expressible in standard Clio with the same value mapping and a suitable filtering condition.

This mapping is underspecified with respect to the target schema. For example, the target area element plays no role in the mapping and is not generated. However, this is not a problem because the area element is optional and, thus, does not need to be generated. The mapping also does not specify how many department elements should be generated. A notion of universal solutions for data exchange was introduced by [7]. Universal solutions satisfy all data dependencies without any extra assumptions. In the case of our mapping example, if only the value mapping is given, then at least two valid solutions exists: one that has one target department with one nested employee element for each source regEmp, and another that has only one target department and as many nested employee elements as source regEmp elements. For our purposes, though, we adopt a minimum-cardinality principle and build as few elements as possible, compatible with the schema constraints. When no builders are given, Clip generates the minimum number of elements necessary for the result to comply with the target schema. For our example, Clip produces the following solution:

```
target---department---employee---@name = Andrew Clarence
                   |---employee---@name = Richard Dawson
                   `---employee---@name = Steven Aiking
```

When builders are used, the proliferation of generated elements is strictly controlled by the builders and the target schema constraints.

**Context propagation.** The mapping in Figure 4 creates a department for each dept and collects all the regEmps (with salary greater than 11000) of the dept as employees of the *corresponding* department. This is performed by drawing a first builder from dept to department (through a build node) and a second builder from regEmp to employee (through another

build node). The build nodes are connected by a context arc to enforce a hierarchy of builders: regEmps are mapped according to the inner builder within the context of a specific dept, fixed by the outer builder (intuitively, by an "outer iteration"). The result is as follows:

```
target---department---employee---@name = Andrew Clarence
     `---department---employee---@name = Richard Dawson
                    `---employee---@name = Steven Aiking
```

Omitting the context arc causes all employees (with sal $>$ 11000) to appear, repeated, within all departments, disregarding the original containment relationships:

```
target---department---employee---@name = Andrew Clarence
     |             |---employee---@name = Richard Dawson
     |             `---employee---@name = Steven Aiking
     `---department---employee---@name = Andrew Clarence
                   |---employee---@name = Richard Dawson
                   `---employee---@name = Steven Aiking
```

**Context propagation tree.** The mapping in Figure 5 is the first example of what cannot be obtained by state-of-the-art tools. This mapping solves the problem we discussed in Section I by specifying a builder and propagating its context *twice*. Namely, department elements are generated by the topmost builder, while project and employee elements are generated considering the current topmost mapping into department.

**Cartesian product and join.** When a build node is reached from two or more source schema nodes, Clip computes the Cartesian product of the source data selected by each builder. Users can add a filtering condition on the label of the build node. If this condition involves two different variables, Clip computes a Join between the source data selected by the build node.

The mapping in Figure 6 combines Projs and regEmps into a flattened list of elements that represents the association of employees to the project they work on (joined on @pid). The topmost build node has no output builder (nothing has to be built at dept granularity); however, a context arc restricts the context of the Cartesian product of Projs and regEmps to nodes within the same dept. The second build node filters the Cartesian product pairs ($p.@pid = $r.@pid), so that the computed operation is a join. This join condition can be entered by the user or can be automatically suggested using the existing referential integrity constraint. The result is:

```
target---project-emp---@pname = Appliances
     |               `---@ename = John Smith
     |---project-emp---@pname = Appliances
     |               `---@ename = Andrew Clarence
     |---project-emp---@pname = Robotics
     |               `---@ename = Jim Bellish
     |---project-emp---@pname = Robotics
     |               `---@ename = Mark Tane
     |---project-emp---@pname = Brand promotion
     |               `---@ename = Richard Dawson
     |---project-emp---@pname = Appliances
     |               `---@ename = Mark Tane
     `---project-emp---@pname = Brand promotion
                     `---@ename = Steven Aiking
```

To better illustrate the roles of these constructs, let us briefly consider two variants of the mapping in Figure 6. If we omit the join condition, then a full Cartesian product is computed and each Proj is associated with all regEmps within their dept. If we also omit the top-level build node, then Clip computes the overall Cartesian product of all regEmps and Projs in the whole document.

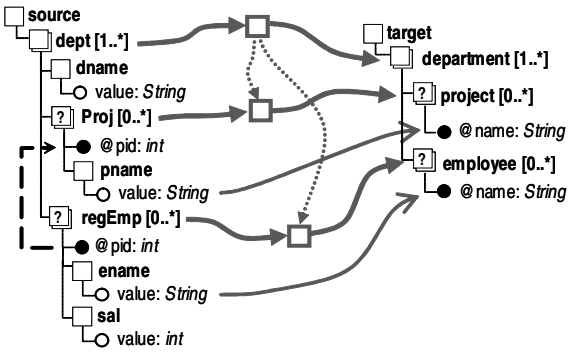



Fig. 5. A more complex Clip mapping

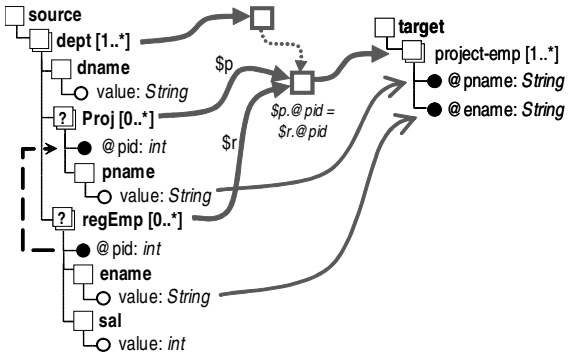



Fig. 6. A join constrained by a CPT

**Grouping.** Group nodes allow users to group source data based on a set of grouping attributes. While regular build nodes produce simple sequences of elements, a group node creates a set of sequences of elements, grouped along the grouping attributes. The cardinality of the result (number of sequences) is given by the number of distinct values of the grouping attributes. The outgoing builder, if any, constructs one target element for each distinct values of the grouping attributes.

The mapping in Figure 7 groups Projs by name. A project is created for each distinct project name, i.e. for each group of homonymous Projs. Each such group is passed as context to

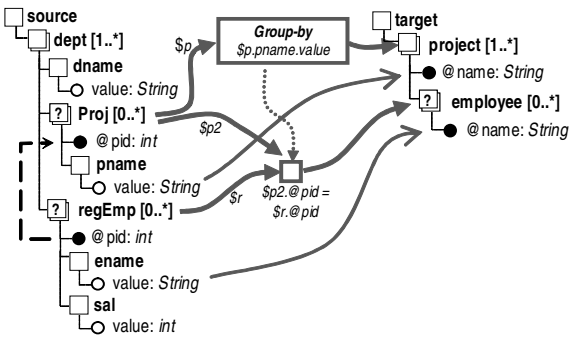



Fig. 7. A mapping with grouping and join

the inner builder, so that regEmps are chosen by comparing their @pids attributes with those of the Projs in one group only. Thus, the generated target will contain as many project elements as there are distinct values of project names in the source instance. Further, the list of employee elements under project will contain the employees that work in that project (independently of the department these employees report to). This construction is necessary because projects are repeated within departments. The result is as follows:

```
target---project---@name = Appliances
     |         |---employee---@name = John Smith
     |         |---employee---@name = Andrew Clarence
     |         `---employee---@name = Mark Tane
     |---project---@name = Robotics
     |         |---employee---@name = Mark Tane
     |         `---employee---@name = Jim Bellish
     `---project---@name = Brand Promotion
               |---employee---@name = Richard Dawson
               `---employee---@name = Steven Aiking
```

Note that values of pnames are legally mapped to the attribute @name of a single project because pname is a grouping attribute and is therefore univocally determined. Non-grouping values have multiple and a-priori different values, and cannot be mapped to the output elements, unless condensed into one value by aggregate functions (as exemplified in the following).

**Other examples.** The mapping in Figure 8 still maps all Proj elements with the same name into a single project. The departments involved in each project appear nested under each target project element. Note that the "innermost" build node takes depts from a higher nesting level in the source with respect to Projs, so as to invert the hierarchy. The result is as follows:

```
target---project---@name = Appliances
     |         |---department---@name = ICT
     |         `---department---@name = Marketing
     |---project---@name = Robotics
     |         `---department---@name = ICT
     `---project---@name = Brand promotion
               `---department---@name = Marketing
```

Last, the mapping in Figure 9 demonstrates aggregate functions by computing for each dept the number of projects and employees and the average salary. Notice that the value mappings tagged $\langle\langle count \rangle\rangle$ can start from multiple elements, too, making an exception to the syntactic rule. The result is as follows:

```
target---department---@name = ICT
     |            |---@numProj = 2
     |            |---@numEmps = 4
     |            `---@avg-sal = 10875
     `---department---@name = Marketing
                  |---@numProj = 2
                  |---@numEmps = 3
                  `---@avg-sal = 20000
```

## III. Validity of Mappings

Not all combinations of value mappings and builders produce valid target instances (i.e. instances that conform to the target schema). We say that a mapping is *valid* if, given any instance of the source schema, the mapping produces a valid instance of the target schema. In this section we discuss several syntactic rules that Clip uses to detect valid mappings. We note

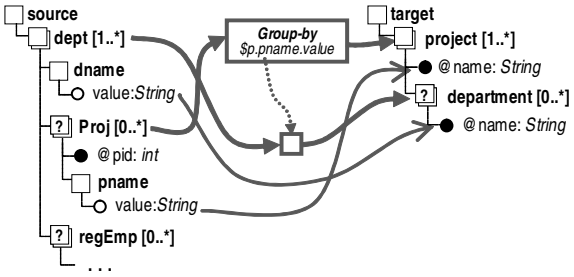



Fig. 8. Inverting the nesting hierarchy

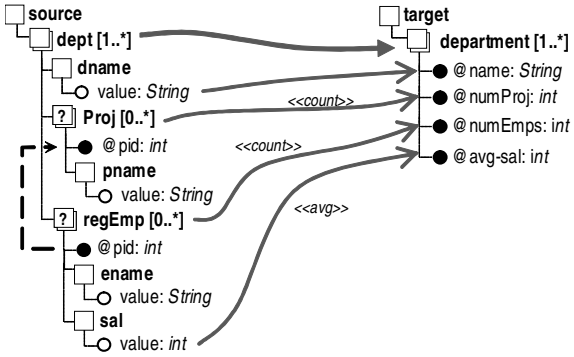
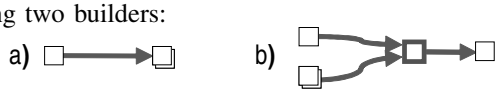



Fig. 9. A mapping with aggregates

that valid mappings require *safe* builders; i.e., builders that allow all source data to be somehow copied to the target side. However, in certain cases, users might need to enter unsafe builders into a mapping. Clip marks these mappings as invalid, but does not restrict the user from entering them.

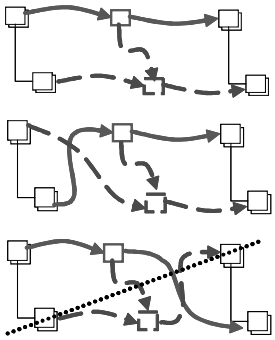
### A. Valid Object Mappings

We say that a builder is *safe* if it goes from more constraining to less constraining schema elements, in terms of the cardinality of those elements. For example, consider the following two builders:




In a), a single element is safely connected to a repeating element. This mapping produces a target singleton set, a valid target instance that "fits" into the target element cardinality. In b), the result of a Cartesian product is connected to a non-repeating element. Since the one of the input is a repeating element, unless we explicitly aggregate the Cartesian product results into a single value, no valid target instance can be generated from this (unsafe) mapping. Safe builders guarantee that every instance bound by the builder can be accommodated in the target.

A CPT is valid if it is a composition of *safe* builders, forming a tree which is topologically *aligned* with the structure of the target schema, i.e. the hierarchy of the build nodes reflects that of the nodes in the target schema reached by the outgoing builders. The examples below illustrate legal and illegal configurations.



*a) **Linear Valid** mapping: the CPT is aligned with both the source and target schemas*

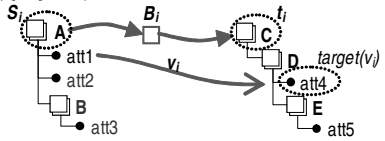
*b) **Inverted Valid** mapping: the CPT is aligned with the target, but not with the source schema.*

*c) **Inverted INVALID** mapping: the CPT is not aligned with the target schema.*

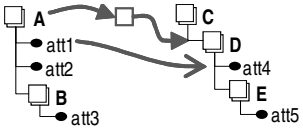
### B. Valid Value Mappings

Source data instances are converted into target data instances using the value mappings connecting leaf nodes (attributes or text values) of the schemas. This data conversion occurs in the *evaluation context* prescribed by the CPT; i.e., each value mapping is driven by a set of build nodes. For each value mapping, we can identify its *driver* by finding the builder node that encompasses the value mappings source and target elements.

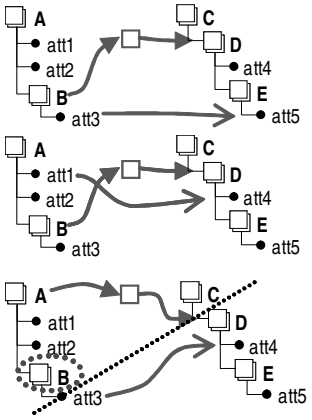
More formally, let $v_i$ be a value mapping between a set of source schema nodes $source(v_i)$ and a target schema node $target(v_i)$. Given a schema node $e$, we define $path(e)$ as the unique set of schema nodes visited if we walk up the schema hierarchy, starting at $e$ and ending at the root node. Starting from $target(v_i)$, we search upward in $path(target(v_i))$ and stop at the first target node $t_i$ that is the target side of a builder. We call this builder $B_i = \langle S_i, t_i \rangle$, where $S_i$ is the set of source schema nodes used by $B_i$, the *driver* of $v_i$. In other words, we find a builder $\langle S_i, t_i \rangle$ such that $\nexists t_k$ with a builder $\langle S_k, t_k \rangle$ in the CPT and $path(t_i) \subset path(t_k)$. The following diagram illustrates these relations using an example. Here, $B_i = \langle \{A\}, C \rangle$ is the driver of $v_i$.




A value mapping $v_i$ is valid if (i) we can find a builder $B_i$ as described above, **and** (ii) for all source nodes $s_v \in source(v_i)$, we can find a source node $s_b \in S_i$ such that $path(s_v) \setminus path(s_b)$ does not contain any repeating source elements. In other words the invalid value mapping uses at least one source-side node that is inside a repeating element that is not bounded by a builder and, thus, Clip does not know how to iterate over that set. The following examples illustrate these concepts:




*a) **Valid:** the atomic attributes involved are direct descendants of the nodes involved in the builder.*




*b) **Valid:** the atomic attribute att3 is a direct descendant of B. Since att5 does not directly descend from D, an E element will be produced, too.*

*c) **Valid:** The builder uses B as its input node. A is in the path to the root from B, and A is also in the path from att1.*

*d) **INVALID:** the nearest ancestor of att3 is B, which is not mapped by any builder.*

The consistency rules above do not apply for value mappings with aggregate functions. Aggregate functions produce a single result out of a set of values with any cardinality. So, the driver of an aggregate value mapping is always valid. If a builder is defined above, it fixes the context of the mapping, restricting the number of values considered by the aggregate node at each iteration; if no builder is specified, the whole document is the scope of aggregation.

## IV. Language Semantics

This section introduces extensions for handling the new Clip's features (builders and aggregates) to the internal languages that we developed in previous works on schema mapping generation [1][2].

### A. Mapping language

We define the semantics of mappings by means of a query-like notation; in the notation, *expressions* and *terms* are defined by $e ::= S \mid x \mid e.l$ and $t ::= e \mid F[e]$; where $S$ is a schema root, $x$ is a variable, $l$ is a label, $e.l$ is a record projection, and $F$ is a function symbol. An *explicit mapping* is represented by a (nested) *tgd* (tuple generated dependency) in the following form:

$$M ::= \forall x_1 \in g_1, \ldots, x_n \in g_n \mid C_1 \rightarrow \\ \exists y_1 \in g'_1, \ldots, y_n \in g'_n \mid (C_2 \wedge M_1 \wedge \ldots \wedge M_n)$$

where $M_1, \ldots, M_n$ are submappings of $M$. Each submapping is itself a mapping; $M$ is an ancestor of $M_1, \ldots, M_n$ and (recursively) of the submappings of $M_1, \ldots, M_n$. Each $x_i \in g_i$ ($y_i \in gi'$) is a *source* (*target*) *generator*. In a source (target) generator, the head of an expression $g_i$ must either be a source (target) schema root, or a variable defined in a source (target) generator of $M$ (in which case it must be some $x_j$ with $j < i$), or in a source (target) generator of an ancestor of $M$. A *source* (*target*) *expression* is an expression over a variable defined in a source (target) generator of $M$ or one of its ancestors. The expression $C_1$ consists of a conjunction of comparisons of type $a_1$ *oper* $a_2$, where $a_1$ is a source expression, $a_2$ is either a source expression or a constant and *oper* is an operator for equality, inequality or membership (in this case, $a_2$ cannot be a constant). The expression $C_2$ has three kinds of formulas. First, it has target conditions: comparisons between target expressions of atomic type and constants. Second, it

has source-to-target conditions: equalities between source and target expressions of atomic type. Finally, it has equalities with functions: equalities of the form $e = F[e_1, \ldots, e_m]$ where $e$ is a target expression of set type, $e_1, \ldots, e_m$ are source expressions, and $F$ is a function.

### *B. Semantics of builders and build nodes*

We now discuss the tgds representing the mappings of Section II.A. Section V will address the algorithmic generation of these tgds.

**Simple mapping.** The mapping of Figure 3 translates to the following simple tgd:

$$\forall\, d \in source.dept, r \in d.regEmp \mid r.sal.value > 11000 \rightarrow \\ \exists\, d' \in target.department, e' \in d'.employee \mid \\ e'.@name = r.ename.value$$

Note that the above tgd does not capture the minimum-cardinality semantics, which requires the construction of only one target department for each source department. As already mentioned, the universal solution would construct a department for each employee; we, nevertheless, enforce minimum cardinality in the generated XQuery, not in the tgd expressions. We discuss our generation of XQuery scripts in Section VI.

**Context propagation.** The mapping depicted in Figure 4 contains a context arc that constrains the scope of the inner mapping within the context of the outer one. This is expressed in our nested tgds with a sub-mapping (in square brackets) as follows:

$$\forall\, d \in source.dept \rightarrow \exists\, d' \in target.department, \\ [\forall\, r \in d.regEmp \mid r.sal.value > 11000 \rightarrow \\ \exists\, e' \in d'.employee \mid e'.@name = r.ename.value]$$

The submapping is correlated to the outer mapping by references to the variables $d$ and $d'$.

**Context propagation tree.** The tgd corresponding to the mapping in Figure 5 has two nested mappings, one for projects and one for employees. Both submappings refer to the same variables $d$ and $d'$ which are bound by the outer mapping:

$$\forall\, d \in source.dept \rightarrow \exists\, d' \in target.department, \\ [\forall\, p \in d.Proj \rightarrow \exists\, p' \in d'.project \mid \\ p'.@pid = p.@pid, p'.@name = p.pname.value], \\ [\forall\, r \in d.regEmp \mid r.sal.value > 11000 \rightarrow \exists\, e' \in d'.employee \mid \\ e'.@name = r.ename.value]$$

**Cartesian product and join.** The tgd for Figure 6 is:

$$\forall\, d \in source.dept \rightarrow \\ [\forall\, p \in d.Proj, r \in d.regEmp \mid p.@pid = r.@pid \rightarrow \\ \exists\, p' \in target.project\text{-}emp \mid \\ p'.@pname = p.pname.value, \\ p'.@ename = r.ename.value]$$

Notice that the outer mapping only restricts the context of the inner one (no element is generated at the outer level); also, two variables in the inner mapping span over the full set of Proj and regEmp (under the current dept) to find pairs with corresponding identifiers.

**Grouping.** To support grouping, we introduce a special Skolem function into the mappings. This function is assigned a target-side *set* variable, implying we will generate a group of values for that target set for every different combination of input parameters to the Skolem function. We represent our Skolem functions as taking two parameters: the grouping context and the grouping attributes. The grouping context is a list of nodes that restrict the scope of the grouping attributes (i.e. a list of target variables already bound in some outer levels). The grouping attributes are the dimensions along which groups are formed. The tgd for Figure 7 is:

$$\exists\, group\text{-}by \\ (\forall\, d \in source.dept, p \in d.Proj \rightarrow \\ \exists\, p' \in target.project \mid \\ p' = group\text{-}by(\perp, [p.pname.value]), \\ p'.@name = p.pname.value, \\ [\forall\, p2 \in p, d2 \in source.dept, r \in d2.regEmp \mid \\ p2.@pid = r.@pid \\ \exists\, e' \in p'.employee \mid e'.@name = r.ename.value])$$

The first argument of group-by is $\perp$ , as Proj elements are unrestrictedly chosen from the whole data set.

**Inverting the hierarchy.** The tgd for Figure 8 is:

$$\exists\, group\text{-}by \\ (\forall\, d \in source.dept, p \in d.Proj \rightarrow \\ \exists\, p' \in target.project \mid \\ p' = group\text{-}by(\perp, p.pname.value), \\ p'.@name = p.pname.value, \\ [\forall\, d2 \in source.dept, p \in d2.Proj \rightarrow \\ \exists\, d' \in p' department \mid d'.@name = d2.ename.value])$$

Again, the first parameter of group-by is $\perp$; indeed, in the examples the grouping node is the root of the CPT. Note that we need a second variable $d2$ and a condition $(p \in d2.Proj)$ in order to compute the inversion.

**Aggregates.** We introduce one function for each type of aggregate and specify the attribute to aggregate as an expression rooted in the variable which restricts the context of aggregation. The tgd for Figure 9 is:

$$\exists\, count, avg \\ (\forall\, d \in source.dept \rightarrow \exists\, d' \in target.department \mid \\ d'.@name = d.dname.value, \\ d'.@numProj = count(d.Proj), \\ d'.@numEmps = count(d.regEmp), \\ d'.@avg\text{-}sal = avg(d.regEmp.sal.value))$$

This example shows how we formalize aggregates independently of grouping, so as to emphasize that they are independent extensions. More specifically, a context of aggregation has to be always specified because hierarchical data structures naturally provide several "structural" aggregation levels, which are "wired" within their nested topology. In Clip, the level of aggregation can be fixed by grouping, but also by ordinary builders. Referring to the example, not all the projects are counted, but only those within a given department, as count is attached to a value mapping whose driver originates from dept.

## V. Mapping Generation

We now describe how to semi-automatically generate Clip mappings by extending techniques used in Clio. Clio generates

tgds using only the schemas and value mappings (mappings between atomic elements) as input. In effects, Clio deduces the builders and the context propagation trees that encompasses the given value mappings. However, as we have mentioned before, Clio cannot automatically create some of the mappings we manually enter in Clip. In this section we discuss how we extended Clio's mapping generation algorithm to cover Clip's mappings.

### *A. Mapping generation in Clio*

Clio's mapping generation algorithm was first described in [1]. Given a source and a target schema, Clio identifies source and target *tableaux*. A *tableau* is a set of schema elements (or attributes) that are semantically related; elements are related, for instance, when they are siblings under the same repeating element (the columns of a table in a relational schema), or when the containing repeating elements are related via a parent-child relationship. Intra-schema constraints (e.g., key/foreign keys) are also used to define tableaux by chasing existing tableaux over the constraints.

Consider for example the source schema in Figure 4. Clio detects three tableaux in that schema: one for the dept set, another for the dept with Proj set, and a larger one that involves all sets in the schema and is computed by chasing over the @pid foreign key. We represent each one of these tableaux using this shorthand notation: {dept}, {dept-Proj}, and {dept-Proj-regEmp, @pid=@pid}.

After the source and target tableaux are computed, Clio creates a matrix source vs. target tableaux. Each entry in this matrix relates a source with a target tableaux and is called a *mapping skeleton*. For each value mapping entered by the user, Clio matches the source and target end-points of the value mappings against all the mapping skeletons and mark as *active* those skeletons encompassing some value mappings. Each active skeleton that is not implied or subsumed by others, emits a logical mapping. For example, for the simple mapping in Figure 4, there are 3 source tableaux (mentioned above) and 2 target tableaux ({department}, {department-project}). This creates 6 mapping skeletons. The entered value correspondence will only match the {dept-Proj-regEmp, @pid=@pid} source tableau and the {department} target tableau. From this skeleton, Clio emits the following tgd expression:

$$\forall\, d \in source.dept, p \in d.Proj, r \in d.regEmp \mid p.@pid = r.@pid \rightarrow \exists\, d' \in target.department, e' \in d'.employee \mid e'.@name = r.ename$$

**Nested Mappings:** A further refinement to this mapping generation algorithm was presented in [2]. Logical mappings generated by Clio are possibly related in that they share part of the source and target expressions. In those cases, mappings can be "nested" inside others, reducing the overall number of mapping expressions.

### *B. Mapping generation in Clip*

To illustrate the problem with Clio's mapping generation consider the mapping in Figure 10. Here Clio would compute the 5 source tableaux and the 2 target tableaux shown in the dependency graph. Assume that only the value mappings from $B$ and $D$ are given as input (i.e., the user did not enter the value mapping from $A$). Clio activates two mapping skeletons: $AB \rightarrow FG$ and $AD \rightarrow FG$. But since the more general skeleton $A \rightarrow F$ is not active, Clio's current algorithm cannot nest the two active mappings. Further, notice that what we want is to compute a Cartesian product between $B$ and $D$ using $A$ as a context. To get Clio to produce something close to this, we first need to add an $ABD$ source tableau (shown as $A(B \times D)$ in Figure 10). With this tableau in place, Clio emits another mapping: $ABD \rightarrow FG$. But the query generated by this mapping pairs all $B$ values with all $D$ values regardless of $A$.

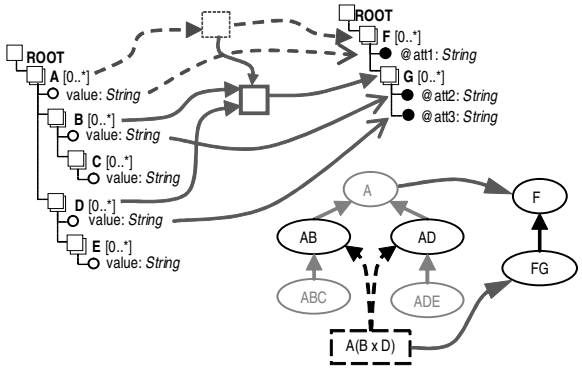



Fig. 10. A generic mapping with its tableaux and dependency graph

Our extension to Clio's nesting algorithm resolves these limitations. Our extension works as follows. First, the nested mappings are computed as usual. Then, we identify all "root" nested mappings (active mappings that are not nested under other mappings). In the case of our example, all mappings are root nested mappings. We then walk up the hierarchy of mapping skeletons, starting from the nested mapping roots, looking for more general mappings that intersect our paths. In the case of our example, this common mapping is $A \rightarrow F$. We mark this mapping as active (regardless of whether it contains value mappings or not) and recompute the nested mappings using the new root mappings.

Consider our first example in which the only active mappings are $AB \rightarrow FG$ and $AD \rightarrow FG$. Our extended algorithm detects $A \rightarrow F$ as a new root nested mapping and nests $AB \rightarrow FG$ and $AD \rightarrow FG$ inside this new root mapping. Nesting mappings has the effect of removing redundant source and target variables, resulting in the following nested expression:

$$\forall\, a \in A \rightarrow \exists\, f \in F \; [\forall\, b \in a.B \rightarrow \exists\, g \in f.G \mid g.@att2 = b.value], \; [\forall\, d \in a.D \rightarrow \exists\, g \in f.G \mid g.@att3 = d.value]$$

In the case of our second example, after the $ABD$ source skeleton is added, Clio outputs $ABD \rightarrow FG$ as an active mapping. It turns out that $ABD \rightarrow FG$ is a sub-mapping of $A \rightarrow F$ (it is not a sub-mapping of $AB \rightarrow FG$ or $AD \rightarrow FG$ because the target side of the mappings is the same.) Our extension will again detect that the more general $A \rightarrow F$

can nest $ABD \rightarrow FG$ inside. The resulting nested mapping captures the Cartesian product with respect to the $A$ values that we wanted:

$$\forall a \in A \rightarrow \exists f \in F \; [\forall b \in a.B, d \in a.D \rightarrow \exists g \in f.G \mid g.@att2 = b.value, \; g.@att3 = d.value]$$

There are two important observations regarding the relation of Clip mappings and Clio mappings. First, Clip's build nodes correspond to Clio's mapping skeletons. For each build node, we look at all its source side builders and match them against the computed source tableaux. If a build node appears in a context propagation tree, we collect all source-side builder arcs and match all of them to a source tableau. If no source tableau is found, we create a new tableau that will cover our source builders and add it to Clio's list. We do the same for the target side of each builder. At the end of this process, we will have identified a source and a target tableau that form the context of our build node. By definition, this tableaux pair is the Clio mapping skeleton that matches the build node.

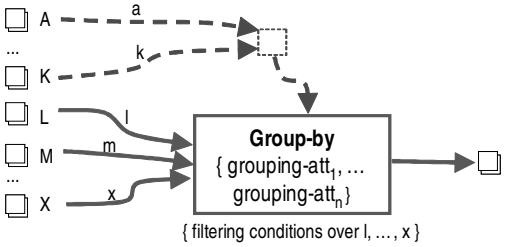
The second observation is that the context propagation tree tells us how to nest the mappings at each build node. In other words, a CPT is a nested mapping. Clip users can rely on our extended nested mapping generation algorithm to automatically compute CPTs for the input value mappings or, users can explicitly enter build nodes and context lines and our algorithm will make sure that the mappings are nested according to the given CPT.

## VI. Translation of mappings into XQuery

The transformation of data instances from the source to the target schema is done by a program generated from the tgds. Since Clip is designed for XML Schema mappings, XQuery is a natural candidate as our transformation language. The algorithm that translates tgds into XQuery is an extension of the code generation algorithms developed for Clio's nested mappings [2]. Here we sketch the general ideas behind the algorithm and concentrate only on the novel and more interesting generation of queries involving grouping and aggregate functions.

The query generation algorithm takes as input a nested mapping $M$ as defined in Section IV-A and produces an XQuery FLWOR expression $F$ as output. Each sub-mapping of $M$ translates into one (nested) FLWOR expression of $F$. $F$ has the following structure: a for clause captures the iteration implied by every universally quantified variable of $M$; a where clause captures the join and filtering predicates in $C_1$; a return clause constructs the XML items for the target schema elements mentioned in the existentially quantified part of the mapping; elements bound to some of the variables defined in the for clauses are copied in the proper position according to the value mappings expressed by the conditions in $C_2$. The sub-mappings of M recursively replicate this structure, so that the scope of quantified variables in the tgds directly translates to that of variables in the nested FLWOR expressions.

It is worth noting that the generated XQuery expressions take into account our *minimum-cardinality assumption*, something not explicitly captured in the tgd expressions. We do this by translating all those elements of the target schema which are not existentially quantified into constant tags and placing such tags in a specific position within the nested FLWOR expressions. More specifically, such constant tags are placed to wrap the FLWOR expression $F$ generated in correspondence with the specific submapping, instead of being placed inside the return clause of $F$; thus, only one element for the whole clause is generated, instead of one for each iteration. In other words, our translation principle is that all the for clauses in the generated FLWOR expressions are pushed as down as possible in the query structure, whenever their nesting level is not enforced by explicit quantification.




**XQuery Grouping expressions.** Consider the above generic grouping build node. The corresponding tgd is the following:

$$\exists \, group\text{-}by \; (\forall l \in \ldots L, \ldots, x \in \ldots X \mid \{filtering \; conditions\} \rightarrow \exists t \in \ldots T \mid t = group\text{-}by([a, \ldots, k][grouping\text{-}attrs]), \ldots$$

Generating an XQuery expression from this kind of tgd expression is a hard task, because the current XQuery standard does not include a grouping statement or clause. We implement the grouping semantics using the existing clauses and functions as follows. First, the for and where clauses are generated in the same manner as with regular build nodes. Then, for each grouping attribute, we create a let clause that computes the distinct values of that attribute in the input data. We then add a for clause that loops over every such distinct value of the grouping attributes. Each iteration over these loops defines the current unique value that is the key value of the group. We then output the desired target elements using the contents of the current group. Any submapping receives the current group as its context. The above generic tgd translates to the following XQuery template:

```
let $context :=
      (for $m in .../M, ..., $x in .../X
       where (: filtering conditions :)
       return <element> {$m} ... {$x} </element>),
    $dim_1 := distinct-values($context/.../@attr1),
     ... ,
    $dim_n := distinct-values($context/.../@attrn)
return
for $d1_val in $dim_1, ..., $dn_val in $dim_n
let $group := (for $x in $context
               where $x/.../@attr1 = $d1_val
               and ...
               and $x/.../@attrm = $dm_val
               return $x )
return
  (: target element, value mappings, submappings :)
```

Consider, for example, the mapping in Figure 7, where

projects are grouped together by their name. The tgd is:

$$\exists\, groupby$$
$$(\forall\, d \in source.dept, p \in d.Proj \rightarrow \exists\, p' \in target.project \mid$$
$$p' = groupby(\perp, p.pname.value),$$
$$p'.@name = p.pname.value)$$

This translates to the following XQuery that builds a `<project>` element for each different project name:

```
<target> {
   let $context :=
         (for $p in source/dept/Proj
          return <element> {$p} </element>),
       $pname_vals := distinct-values(
                        $context/Proj/pname/text())
   for $pname_val in $pname_vals
   let $group := (for $p in $context/Proj
                  where $p/pname/text()=$pname_val
                  return $p )
   return <project name = {$pname_val} />
} </target>
```

As for aggregates, the availability of native XQuery aggregate functions facilitates the translation with respect to the case of grouping. The tgd for the mapping in Figure 9 (limited to the first two value mappings) is:

$$\exists\, count$$
$$(\forall\, d \in source.dept \rightarrow \exists\, d' \in target.department \mid$$
$$d'.@name = d.dname.value,$$
$$d'.@numProj = count(d.Proj))$$

This translates to the following XQuery statement:

```
<target> {
   for $d in source/dept
          return <department
                    name = {$d/dname/text()}
                    numProj = {count($d/Proj)} />
} </target>
```

Notice that the information about the aggregation context carried by the *count* function in the tgd is used to determine the starting point of the path expression argument of the XQuery count function (namely, variable $d).

## VII. Evaluation of Clip and Conclusion

Clip's current implementation includes two components: a GUI for mapping expression and a translator, which produces tgds corresponding to the mapping. The translation of tgds into XQuery is currently ongoing, but this task was already performed by some of the authors in the context of Clio and of its nested version, and it will not introduce significant technical challenges. The GUI interface has been designed by reusing our experience gained in XQBE [10] and Clio [1], by aiming at the best balance between ease of use, expressive power, and effectiveness.

The main "performance" metric for Clip is the number of legal Clip mappings that can be generated for a given set of value mappings. In particular, we can compare the "flexibility" of the mapping interface of Clio and Clip. By flexibility we mean how many different (meaningful) mappings the tool allows us to visually construct. Again, the "meaningfulness" of mappings depends a lot on the particular data integration scenario and it is difficult to enumerate all such possible mappings. Instead, Table I shows a lower-bound of how many more different meaningful mappings we could draw using Clip starting from the same value mappings. This exercise was done using 3 published examples of Clio mappings plus one of the mappings we used in this paper. The first column shows the source of the mapping example, the second column shows how many value mappings are involved in this example, and the third column shows how many more different nested mappings we could create when compared to nested mappings generated by Clio.

TABLE I
Flexibility of Clip

| Example (Source) | Value mappings | Extra meaningful mappings with Clip |
|---|---|---|
| Figure 1 in [2] | 7 | 4 |
| Figure 3 in [2] | 4 | 1 |
| Figure 1 in [1] | 3 | 1 |
| Figure 1 (this paper) | 2 | 4 |

In our future work, the GUI will be augmented by including schema matching tools, i.e. tools suggesting related elements and structures within two complex source and target XML schemes, and by adding filters highlighting some of the lines and of the source and target structures, providing a clear rendering of the lines in the middle [11]; these view mechanisms allow users to concentrate on a portion of the schemas at a time. These additions will help users work with large schemas, as otherwise they could be overwhelmed by schema complexity and by the number of lines from source to target.

## Acknowledgements

Paolo Papotti was partially supported by an IBM Faculty Award grant and Alessandro Raffio was partially supported within a joint study agreement between IBM and Politecnico di Milano.s

## References

[1] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin, "Translating Web Data," in *VLDB*, 2002, pp. 598–609.
[2] A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa, "Nested Mappings: Schema Mapping Reloaded," in *VLDB*, 2006, pp. 67–78.
[3] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm, "Applying Model Management to Executable Mappings," in *SIGMOD*, 2005, pp. 167–178.
[4] A. Bonifati, E. Q. Chang, T. Ho, V. S. Lakshmanan, and R. Pottinger, "HePToX: Marrying XML and Heterogeneity in Your P2P Databases," in *VLDB*, 2005, pp. 1267–1270.
[5] P. A. Bernstein and S. Melnik, "Model management 2.0: manipulating richer mappings," in *SIGMOD*, 2007, pp. 1–12.
[6] M. Lenzerini, "Data Integration: A Theoretical Perspective," in *PODS*, 2002, pp. 233–246.
[7] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, "Data Exchange: Semantics and Query Answering," in *ICDT*, 2003, pp. 207–224.
[8] R. Fagin, P. Kolaitis, L. Popa, and W.-C. Tan, "Composing Schema Mappings: Second-Order Dependencies to the Rescue," in *PODS*, 2004, pp. 83–94.
[9] C. Yu and L. Popa, "Semantic Adaptation of Schema Mappings when Schemas Evolve," in *VLDB*, 2005, pp. 1006–1017.
[10] D. Braga, A. Campi, S. Ceri, and A. Raffio, "XQBE: a visual environment for learning XML query languages," in *SIGMOD*, 2005, pp. 903–905.
[11] G. G. Robertson, M. P. Czerwinski, and J. E. Churchill, "Visualization of Mappings Between Schemas," in *SIGCHI*, 2005.