Spreadsheet-based Complex Data Transformation

Regis Saint-Paul[1]
Hung Vu[2] Ghazi Al-Naymat[2] Boualem Benatallah[2]

[1] CREATE-NET, Italy
`regis.saint-paul@create-net.org`
[2] University of New South Wales, Australia
{`vthung,ghazi,boualem`}`@cse.unsw.edu.au`

Technical Report
UNSW-CSE-TR-0919
October 2009

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Spreadsheets are used by millions of users as a routine all-purpose data management tool. It is now increasingly necessary for external applications and services to consume spreadsheet data. In this paper, we investigate the problem of transforming spreadsheet data to structured formats required by these applications and services. Unlike prior methods, we propose a novel approach in which transformation logic is embedded into a familiar and expressive spreadsheet-like formula mapping language. All transformation patterns commonly provided by popular transformation languages and mapping tools are supported in the language. Consequently, the language avoids cluttering the source document with transformations and turns out to be helpful when multiple schemas are targeted. Furthermore, the language supports the generalization of a mapping from instance-level to template-level element. This enables the language to transform a large number of naturally occurring spreadsheets, which cannot be effectively handled by the alternative approaches. We implemented a prototype and evaluated the benefits of our approach via experiments in two real applications.

1 Introduction

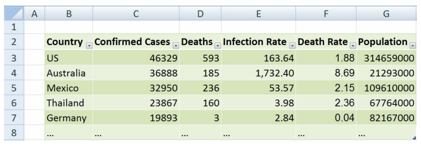
Spreadsheets are ubiquitous tools used for the storage, analysis and manipulation of data [24]. There are several reasons for their popularity. Spreadsheet-based data management offers important flexibility in data formatting over a tabular grid [6]. Spreadsheets do not impose many constraints regarding the data layout. Data can be organized according to subjective importance, preferences, and styles (e.g., by placing important data in the top-left corner or placing related elements of data next to each other). Furthermore, spreadsheets offer a simple, but effective formula language using spatial relationships that shield users from the low-level details of traditional programming [13]. To use the language, a user only needs to master two concepts, namely cells as variables and functions for expressing relations between cells. Consequently, spreadsheets are widely used by knowledge workers (e.g., accountants, project managers, programmers) who play a key role in critical enterprise activities [12].

Given the ubiquity and utility of spreadsheets, it is increasingly necessary to allow data stored in spreadsheets to interact with external applications and services [18]. There has been a proliferation of online spreadsheet-like applications including Google Spreadsheets [3], Excel Web App [2], and Zoho Spreadsheet [5]. To enable other applications to consume or generate spreadsheet data, some of these applications provide Web service interfaces (APIs). The authors in [12] report that spreadsheets often serve as hubs for organizing and manipulating information, which is later transferred to other services for archiving or processing.

In this paper, we consider the problem of transforming spreadsheet data into structured formats required by external applications and services. We believe that facilitating interoperation between spreadsheets, applications and Web services will profoundly improve the effectiveness of information and services management in a variety of domains. However, the problem is challenging because of the nature of spreadsheets: (i) the data they contain does not conform to a predefined schema; (ii) there may be a mismatch between the organization of spreadsheet data and the structure expected by an external application. For example, the spreadsheet in Figure 1.1(a) contains data arranged in a table while the schema used by a bar chart (Figure 1.1(d)) consists of a list of labels, each comprising a list of bars.

Main stream solutions to data transformation rely on specifying mappings between elements of the source and target schemas to transform a source instance to the target format [9]. However, there are many cases in which the schema of the source instance is unknown and transformation is performed directly from the source instance to the target format. For example, end-user visualization websites [26, 4, 11] let users upload a data set (i.e., a source instance) and assist them in transforming it to the format required by a given visualization type (e.g., chart, map, and timeline) with its own target schema.

There are three main existing approaches. The first approach, namely schema-based, allows users to specify schemas of spreadsheets via a layout specification language [15], and then transformation can be performed at *schema level* using either low-level transformation languages (e.g., XSLT/XQuery), or high-level mapping tools, such as Clio [19, 10], Clip [20], and Altova MapForce [1]. However, users must learn a new language, e.g., by creating correspondences between the source and target elements and annotating those correspondences



	A	B	C	D	E	F	G
1							
2		Country	Confirmed Cases	Deaths	Infection Rate	Death Rate	Population
3		US	46329	593	163.64	1.88	314659000
4		Australia	36888	185	1,732.40	8.69	21293000
5		Mexico	32950	236	53.57	2.15	109610000
6		Thailand	23867	160	3.98	2.36	67764000
7		Germany	19893	3	2.84	0.04	82167000
8		...	...	...	...	...	...

(a)

```
PieChart
  Pies[ ]
    Pie
      Name
      Value
```

(b)

```
ScatterPlot
  Dots[ ]
    Dot
      X
      Y
      Label
      Size
```

(c)

```
BarChart
  Labels[ ]
    Label
      Name
      Bars[ ]
        Bar
          X
          Y
```

(d)

Figure 1.1: (a) The Swine Flu data set; (b) *Pie Chart* schema; (c) *Scatter Plot* schema; (d) *Bar Chart* schema

with one or more unfamiliar functions (e.g., functions of XSLT/XQuery or .NET Framework) in the case of mapping tools [23]. This flowchart-like mapping interface is typically cluttered when schemas are large and mappings are complex [22]. On the contrary, spreadsheet users are familiar with formulas and an incremental approach to building applications with instant feedback [13].

The second approach, namely column-based, enables users to specify simple mappings between spreadsheet columns and target attributes (atomic elements) via drag-and-drop operations [21, 8]. This approach requires direct correspondences between spreadsheet column contents and the values of target attributes. For example, after dragging attribute `Y` of scatter plot (Figure 1.1(c)) onto source column `Infection Rate` (Figure 1.1(a)), the values of the column are *copied* to values of the attribute. Such straightforward correspondences, however, are unlikely when the target application and the spreadsheet have been developed independently. For instance, while the infection rate is expressed per million in the source, the target scatter plot expects a rate of per one hundred thousand. To correct this issue, the source spreadsheet must be modified, e.g., the values of column `Infection Rate` must be changed.

The third approach, visualization specific, supports simple mappings in the context of visualization [26, 11]. It compares atomic types of source columns and target attributes of visualization types to suggest mappings to users. For example, to visualize the Swine Flu data set (Figure 1.1(a)) using Pie Chart (Figure 1.1(b)), one of five candidate columns `Confirmed Cases`, `Deaths`, `Infection Rate`, `Death Rate`, `Population` can be mapped to attribute `Value` since all of them have the same type float. Similar to the column-based approach, source column data directly corresponds to the values of target attributes.

In summary, all three approaches suffer from *at least* one of the following drawbacks: (i) Existing programming experience of spreadsheet users (e.g., spreadsheet formulas with instant feedback at each step) is not leveraged; (ii) The transformation may be tedious to accomplish since it can involve multiple manipulations on the spreadsheet. It can also clutter the original organization of the spreadsheet with transformations. This issue may be aggravated if users need to interact with several different target applications, e.g., if requesting quotations by interacting with various supplier Web services or if targeting several visualizations as depicted in Figure 1.1; (iii) There is no reuse support of a mapping for multiple spreadsheets with similar structure, which makes the transformation of these spreadsheets very time-consuming.

1.1 Contributions

To address these issues, we propose a novel approach called TranSheet, which enables users to perform mappings via a familiar and expressive spreadsheet-like formula language. The main contributions of this paper are as follows:

- A spreadsheet-like formula language is designed for specifying mappings between spreadsheet data and the target schema. In terms of expressiveness, we demonstrate that all transformation patterns commonly provided by popular transformation languages (e.g., XSLT/XQuery) and mapping tools [7] are supported in the language via spreadsheet formulas and functions. This enables the language to avoid cluttering the spreadsheet with transformations and it turns out to be helpful when multiple schemas are targeted (Section 3).
- The proposed language supports the generalization of a mapping from instance-level to template-level element allowing the mapping to be applied to multiple instances with similar structure. Frequently used formatting features of spreadsheets are exploited to generalize mappings (Section 4).
- We use *tuple generating dependencies* (*tgds*) [9], a widely used schema mapping formalism, to describe the semantics of TranSheet. We introduce a collection of new functions to tgd expressions. We then extend a previous query generation algorithm [20] to generate executable queries for these functions (Section 5).
- We provide GUI-based transformation utilities, including drag-and-drop operations and form-based wizards, that allow users to specify mappings graphically, rather than writing complex formulas from scratch. Mapping formulas are automatically generated as results of drag-and-drop operations or graphically form-based manipulations (Section 6).
- Finally, we implemented a prototype and evaluated the expressiveness and mapping generalization of TranSheet in two real applications. The experimental results show that our language is expressive and flexible enough to support numerous practical spreadsheet-based transformation scenarios (Section 7).

1.2 Application scenarios

Since a significant amount of the world's data is stored in spreadsheets, we believe TranSheet has numerous applications in data exchange for both desktop and Web-based environments.

Business users can use TranSheet to interact with the enterprise applications of their organization, such as CRM and ERP, to get information for analysis. For instance, to analyze sales performance using a spreadsheet, a salesperson may use TranSheet to interact with the services exposed by Salesforce CRM to retrieve notifications of new sales leads.

TranSheet can be used as a transformation plug-in for end-user visualization websites, such as ManyEyes [26, 4] and Google Fusion Tables [11]. TranSheet enables users to map a dataset to different visualization types while keeping the dataset unmodified. Regarding what currently offers by these websites, users must perform multiple manipulations on source documents as well as maintain many versions of them, each for a visualization. This makes transformation cumbersome and laborious.

A small retailer might use TranSheet to request quotations from several big online suppliers such as Amazon, Ebay, and PriceGrabber. These quotations may be stored in a spreadsheet containing product information for making reports and selecting the most appropriate price - regardless of differences in the various suppliers' Web service interfaces.

2 Data Model

The problem of exporting data from a spreadsheet to an XML document is a particular instance of the source-to-target data transformation problem that has been the subject of previous research in the area of data integration. It consists of expressing a mapping between the source and target data model. In this section, we present our modeling of spreadsheet documents in Section 2.1 and the data model we use to represent the target XML schema in Section 2.2.

2.1 Spreadsheet data model

A spreadsheet S (where S, from now on, will stand indifferently for Source or Spreadsheet) is modeled as a bi-dimensional matrix of cells. (Note that spreadsheet containing multiple worksheet may be modeled using an additional dimension but extension of this work to such a model is trivial.)

Each cell is identified by its coordinates. We use the coordinate $\langle x, y\rangle$ to denote the cell corresponding to column x and row y. Following spreadsheet conventions, cell coordinates are numbered starting from 1 and can also be denoted using capital letters for column numbering followed by numbers for rows numbering. For example, cell $\langle 1, 5\rangle$ can also be denoted as cell A5. Cell A1 is the upper-leftmost cell. A rectangular subset of cells is called a *range* and is denoted by its upper-leftmost and lower-rightmost cells separated by a colon. For example, D3:E13= $\{\langle x, y\rangle | 4 \leq x \leq 5, \quad 3 \leq y \leq 13\}$. A range $\langle x_1, y_1\rangle$:$\langle x_2, y_2\rangle$ always verifies $1 \leq x_1 \leq x_2$ and $1 \leq y_1 \leq y_2$. Note that if $x_1 = x_2$ and $y_1 = y_2$, we have $\langle x_1, y_1\rangle$:$\langle x_2, y_2\rangle$=$\langle x_1, y_1\rangle$, that is, a cell can be seen as a range of one row and one column. The spreadsheet environment also allows for the definition of ranges of noncontiguous cells. These are denoted by separating each cell or range of cells with a comma[1]. A noncontiguous range is a simple list of cells, where cells are listed in reading order.

Each cell has an associated typed atomic value. τ_a represents the set of atomic types supported by the spreadsheet environment to describe the content of cells. We consider $\tau_a ::= empty|int|float|string|datetime$. τ_a could be of course extended to support additional types (e.g., charts or images [17]). The type *empty* represents the special case of an empty cell.

The spreadsheet data model has therefore the very special characteristic of being essentially "visual": its structuring occurs through a combination of spatial elements (i.e, layout on the grid) and graphical elements (i.e, font style, borders). This is different from other data models such as XML or the relational

[1]We adopt here the MS Excel convention; For instance, OpenOffice uses a semi-colon to the same effect.

	A	B	C	D
1	**0042**			
2	*Ford Prefect*	*Addison*	*Sydney*	*NSW*
3	75.64	150	Beer	
4	5.26	2	Towel	
5	4.32	1	Babel Fish	
6				
7	**0525**			
8	*Arthur Dent*	*Evans*	*Melbourne*	*VIC*
9	2.75	1	Towel	
10		...		

(a) Source spreadsheet with hierarchical representation of orders

```
QuoteRequest
   Account
      Login ="MyLogin" {MyLogin}
      Password ="MyPass" {MyPass}
   Orders [ ] (1 item)
      Order
         Id =A1 {0042}
         ShipTo
            FirstName =left(A2,search(' ', A2)) {Ford}
            LastName =right(A2,len(A2)-search(' ',A2)) {Prefect}
            Address =concatenate(B2,' ',C2,' ',D2) {Addison...}
         OrderDetails [ ] (3 items)
            OrderLine
               Quantity =B3:B5*10 {1500, 20, ...}
               ProdName =C3:C5 {Beer, Towel, ...}
               Price =round(A3:A5,0) {76, 5, ...}
```

(b) Schema view with mappings for order 0042

Figure 2.1: A spreadsheet and its mapping specification

data model, where the structuring can be referred to in terms of the symbols (e.g. XML elements, table names and attributes).

2.2 Target Data Model

As mentioned before, we focus on exportation of spreadsheet data to XML. For this purpose, we use a slight variation of the XML data model proposed in [28]. Several variations of this model have also been used in earlier work (e.g., [19]).

The purpose of this data model is to retain the essential features of the hierarchical XML data model while abstracting away representation details such as whether a label is an element or an attribute.

We model a schema as a set of typed labels. The set τ_t of label types is defined as follows: $\tau_t ::= \tau_a \mid SetOf[l_i{:}\tau_t^i] \mid Rcd(l_1{:}\tau_t 1, \ldots, l_n{:}\tau_t^n)$. In this notation, l_is are label names and τ_t^is are their respective types. The symbol $SetOf$ represents repeating elements of an XML schema as an unordered set of any number of the same label, and Rcd represents tuples (i.e., collection of orderded label-value pairs)

For presentation in the user interface, we use an equivalent notation illustrated in Figure 2.1(b), where indentation is used to denote children labels or a label of type Rcd or $SetOf$. The Rcd construct is implicit and the symbol [ ] is used to denote the $SetOf$ construct. Consequently, only atomic types need to be denoted.

In the remainder of this paper, we assume that atomic types τ_a are identical in both the spreadsheet and the XML worlds. In reality, atomic type systems may differ (e.g., XML Schema allows to specify domain restrictions on simple types). These differences may lead to transtyping violation when assigning a value of the spreadsheet to a given label (e.g., an integer value may be out of the range expected in the target document). When a cell value in the spreadsheet

Types	**Transformation patterns**
Value mapping	*Copying*, *Derivation*, *Constant Value Generation*, *Merging*, *Splitting*
Structural mapping	*Nesting*, *Filtering*, *Sorting*, *Grouping with aggregation*, *Join and Cartesian product*, *Union/Intersect/Minus*, *Branching*

Table 2.1: Transformation patterns of TranSheet

cannot be transtyped into a value compatible with type expected in the target schema, a feedback is provided to users.

3 Formula Mapping Language

While atomic labels hold actual values, structural labels control the structural information of schema. Thus, we consider two kinds of mappings, namely *value mapping* (map one or more cells to an atomic label) and *structural mapping* (map a range of cells to a $SetOf$ label). In this section, we demonstrate via examples that all common transformation patterns provided by popular transformation languages and mapping tools are supported in our language (Table 2.1).

3.1 Value mapping

Copying

This pattern simply copies a cell value to a target value of a label. For example, in Figure 2.1(b), we have the following mappings:

- `Id`=A1 copies the value corresponding to A1 to the value of `Id`. Instant feedback for the mapping is provided in the curly brackets {*0042*} adjacent to label `Id`.
- `ProdName`=C3:C5 copies three values of cells C3, C4, and C5 to the values of label `ProdName`.

In the last example, a *mono-dimensional range* expression is associated to an atomic label. In spreadsheet programming, range expressions are only used as function parameters (e.g., for computing the total sum of a collection of cells). TranSheet, however, leverages the familiarity users have with the range notation to conveniently express mappings of schema labels with cell collections called *range formulas*. A range formula is valid only for an atomic label that has a $SetOf$ label as ancestor. To show that the `OrderDetails` label allows repetition of label `ProdName`, the text "*(3 items)*" (number of label `ProdName`) is displayed as an additional metadata.

Special care needs to be taken for range formulas associated with atomic labels. Suppose that instead of the mapping shown in Figure 2.1(b), the user inputs the mapping formulas `Quantity`=B3:B17 (15 `Quantity` items) and `ProdName`=C3:C5

(3 `ProdName` items). Taken together, these two mapping formulas violate the schema constraint which states that the target document should contain the same number of `Quantity` and `ProdName` labels. In this situation, the metadata associated with the label `OrderDetails` shows the following warning message:

`OrderDetails` [ ] **Warning**– *Number of items should be the same for all children labels*

TranSheet generates a target document even in the presence of warnings. In our example, only three `OrderDetails` items would be generated, since only three are such that all their children labels have definite values.

It should also be noted that ranges need not be contiguous or even follow the same direction (i.e., column or row). For instance, the pair of mapping formulas `Quantity`=A2:A6 and `ProdName`=F13:J13 are valid and express that the five values for `Quantity` label are found in a same column, while the five values for `ProdName` label are found in a same row.

Constant Value Generation

In some special cases, a constant must be copied to a target value where the constant value is independent of the source. For example, in Figure 2.1(b) the mapping formulas `Login`="MyLogin" and `Password` ="MyPass" associate constants to the values of `Login` and `Password`, respectively.

Derivation

This kind of mapping allows users to use one or more Excel-like functions on strings (e.g., *upper*, *lower*, *trim*), numbers (e.g., "+", "*", "-", "/", *abs*, *ceiling*, *round*), and dates (e.g., *date*, *time*, *hour*) to bring the format of a cell value to the required format of a target value. For example, in Figure 2.1(b):

- Mapping `Quantity`=B3:B5*10 uses a range formula to state that values of `Quantity` correspond to three values in the spreadsheet multiplied by 10.
- Mapping `Price`=round(A3:A5,0) rounds values in the range A3:A5 to the nearest integers and copies to values of `Price`.

Merging

This pattern merges multiple cell values into one value of a target label. For example, in Figure 2.1(b), the mapping formula `Address`=concatenate(B2," ",C2," ",D2) merges the values of three cells B2, C2, D2, which contain information on street, city, state, into the value of label `Address` with delimiters whitespace " " using Excel-like function *concatenate*.

Splitting

This kind of mapping is used when splitting one cell value into one or multiple target values. For example, mapping formulas `FirstName`=left(A2, search(" ", A2)) and `LastName`=right(A2, len(A2)-search(" ",A2)) in Figure 2.1(b) split the value of cell A2, which contains information on customer name, into the values of labels `FirstName` and `LastName` according to the whitespace " " using Excel-like functions *left*, *search*, *right*, and *len*.

	A	B	C	D	E
1	*OrderId*	*FirstName*	*LastName*	*ProdName*	*Quantity*
2	0042	Ford	Prefect	Beer	150
3	0042	Ford	Prefect	Towel	2
4	0042	Ford	Prefect	Babel Fish	1
5	0525	Arthur	Dent	Towel	1
6	0525	Arthur	Dent	Tea Bags	20
7			...		

(a) Tabular representation of orders

```
QuoteRequest
  Orders [ ] =A2:E50 (49 items)
    Order
      Id {0042, 0525, ...}
      ShipTo
        FirstName {Ford, Arthur, ...}
        LastName {Prefect, Dent, ...}
      OrderDetails [ ]
        OrderLine
          ProdName {{Beer, Towel, ...}, ...}
          Quantity {{150, 2, ...}, {1, 20, ...}, ...}
```

(b) Mapping specification

Figure 3.1: A tabular representation of orders and its corresponding transformation

3.2 Structural mapping

We first provide an overview about structural mappings and then we present transformation patterns at structural level.

Formula inheritance

When using a structural mapping $l_s = f$, formula f is interpreted in terms of mapping formulas associated with the atomic children labels of structural label l_s. To intuitively illustrate this formula inheritance, the structural mapping `OrderDetails`=B3:C5 in Figure 2.1 is interpreted in terms of lower level mapping formulas in the following two steps:

$$\texttt{OrderDetails}=\text{B3:C5}$$
$$\Downarrow$$
$$\texttt{OrderLine}=\text{B}\langle i\rangle\text{:C}\langle i\rangle, \quad 3 \le i \le 5$$
$$\Downarrow$$
$$\texttt{Quantity}=\text{B}\langle i\rangle, \texttt{ProdName}=\text{C}\langle i\rangle, \quad 3 \le i \le 5$$

As can be seen, by using formula inheritance, children atomic labels `Quantity` and `ProdName` of label `OrderDetails` obtain values from columns B3:B5 and C3:C5, respectively.

Defaults of structural mapping

Figure 3.1 presents a simple structural mapping `Orders` =A2:E50, where several orders are organized in a single denormalized table (nested table). The interpretation process, as presented so far, uses certain defaults: (i) TranSheet assumes that data is organized in a table with attributes as columns and tuples as rows; (ii) TranSheet also takes advantage of the same ordering of columns in the source spreadsheet and target atomic labels (e.g., quantity comes "before" product name in both the spreadsheet and the target schema).

These defaults can be overridden: the first by using function *transpose* to indicate that a table is represented with attributes as row and tuples as column; the second by the *schema restructuring* features described in Section 3.3.

Nesting

The mapping illustrated in Figure 3.1 is potentially ambiguous since two distinct target instances are possible: either (i) grouping products per order, as could be expected, or (ii) mimicking the data organization of the spreadsheet document with as many order labels as there are products. In this example, both generated documents satisfy the mapping specification as well as the target schema. However, the document where products are grouped per orders is often desirable [19, 10]. By default, the target document is nested according to order identifier, first name, and last name.

Filtering

This kind of structural mapping allows users to select data from a set of source tuples according to specific filtering conditions. For example, in Figure 3.1 the user wants to select orders from the source whose product name is equal to "Towel" and quantity is greater than 1. This can be obtained by associating a filtering predicate *filterexp* to the structural mapping `Orders` = A2:E50. *Filterexp* is typically a combination of Excel-like logical functions *AND*, *OR*, and *NOT*. For example, the following mapping is employed:

`Orders` =A2:E50[AND(D2:D50="Towel", E2:E50>1)]

Sorting

This kind of structural mapping allows users to sort tuples in the source spreadsheet according to values of columns. It is defined via function *sort(column1, order1, column2, order2,...)*, where *column*$_i$ is a column specified by a one-dimensional range, and *order*$_i$ is the corresponding sorting order of *column*$_i$ with value "ascending" or "descending". For example, the list of orders in Figure 3.1 can be sorted according to the product name in ascending order, and then according to the quantity in descending order as follows:

`Orders` =A2:E50[sort(D2:D50, ascending, E2:E50, descending)]

Grouping with aggregation

In this example, the user wants to group the source spreadsheet in Figure 3.1(a) by order identifier, first name, and last name. Excel-like aggregate functions (e.g., *count*, *sum*, *avg*, *min*, and *max*) can then be used together with grouping to calculate values for product name and quantity in each group. The target schema to be mapped is:

```
Target
  Orders [ ] =A2:E50[groupby(A2:A50, B2:B50, C2:C50)]
    Order
      Id {0042, 0525, ...}
      FirstName {Ford, Arthur, ...}
      LastName {Prefect, Dent, ...}
      ProdName =count(D2:D50) {3,2, ...}
      Quantity =max(E2:E50) {150,20, ...}
```

The following mappings are employed:

- `Orders` =A2:E50[groupby(A2:A50, B2:B50, C2:C50)] where function *groupby(column1,column2, ...)* groups a set of tuples according to values in columns $column_1$, $column_2$, and so on. This structural mapping is then refined at leaf level on atomic labels `ProdName` and `Quantity`.
- `ProdName` =count(D2:D50) counts the number of products for each order.
- `Quantity` =max(E2:E50) finds the maximum value in the set of quantities associated with an order.

Join

Suppose that the source spreadsheet in Figure 3.1(a) is divided into tables A2:C6 and D2:F3 where table A2:C6 contains order details and table D2:F3 contains customer information:

	A	B	C	D	E	F
1	*OrderID*	*ProdName*	*Quantity*	*OrderID*	*FirstName*	*LastName*
2	42	Beer	180	42	Ford	Prefect
3	42	Towel	2	525	Arthur	Dent
4	42	Fish	1			
5	525	Towel	1			
6	525	Teabags	20			

The two above tables are joined according to order identifiers and mapped to the target schema in Figure 3.1(b). This can be achieved via function *join(table1, table2, joincondition)* where *table1* and *table2* are two tables defined by two-dimensional ranges and *joincondition* is the optional condition to join two tables. We have the following mappings:

`Orders` =join(D2:F3, A2:C6, D2:D3=A2:A6)

`Id` =D2:D3; `FirstName` =E2:E3; `LastName` =F2:F3

`OrderDetails` =B2:C6

When *joincondition* is missing, a full Cartesian product is computed between two tables.

Union/Intersect/Minus

The *union* function allows users to union two tables with the corresponding signature *union(table1, table2,...)* where *table1*, *table2*,... are tables to be unioned. By default, union is duplicate-eliminating. For example, the user wants to union two the following two tables with duplicate removal and map them to the target schema shown in Figure 3.1(b):

	A	B	C	D	E	F
1	42	Ford	Prefect	Beer	150	
2	42	Ford	Prefect	Towel	2	
3	42	Ford	Prefect	Babel Fish	1	
4						
5	525	Arthur	Dent	Towel	1	
6	525	Arthur	Dent	Tea Bags	20	

The following structural mapping is associated with the label `Orders`: `Orders` =union(A1:E3, A5:E6). Other set operators intersect and minus can be specified via functions *intersect(table1, table2,...)* and *minus(table1, table2,...)*, respectively.

Branching

This kind of mapping allows users to map different sets of data to a target label depending on the outcome of a preset condition. It is specified via Excel-like function *if(condition, value_if_true, value_if_false)*. For example, suppose that there is an additional attribute named `Description` located right below element `Quantity` of the target schema in Figure 3.1(b) to indicate whether the quantity of an order item is small or large. The following mapping is used together with mapping `Orders =A2:E50`: `Description` =if(E2:E50>20, “large”, “small”)

That is if the quantity of an order item is greater than 20, then attribute `Description` is assigned “large”; otherwise it is associated with “small”. Only the order item with quantity 150 in the second row in Figure 3.1(a) is assigned “large”.

Composition and refinement of mappings

A *complex transformation* is typically composed of multiple patterns of structural and value mappings. For example, the user can perform sorting and filtering at structural level, and derivation and merging at leaf level. Additionally, a formula specified on a label may collide with a formula inherited from one of its ancestor labels. Regarding mapping `Orders`=A2:E50 in Figure 3.1, the user can see that, by inheritance, the values associated to label `ProdName` are {*150, 2, ...*} which is a copy of the quantity column of the spreadsheet. However, the user expects that values of label `Quantity` should correspond to values of this column multiplied by 10. In such situation, the user can specify mapping formulas at structural labels and refine them at a lower level (e.g., leaf level). These will have precedence over the values derived by the automated mechanism. For instance, mapping `Quantity`=E2:E50*10 is used to correct the problem. An interactive refinement can be performed until the user is satisfied with the example values displayed as a mapping preview.

3.3 Target schema restructuring

We have seen that TranSheet relies on the order and nesting of elements in the schema view in order to interpret the mapping specification. This feature allows a mapping formula to be very concise in the case where the spreadsheet and the target schema have a matching organization. However, it is a common occurrence that target schemas, which are defined externally, do not coincide with the spreadsheet organization. Our solution is to allow users to organize a view of the target schema by a set of *rearrangement* operations. By rearranging the schema view, users do not modify the underlying target schema; they merely specify, in a graphical way, how spreadsheet data are organized and, implicitly, how mapping formula should be interpreted.

Isomorphic view rearrangement

Isomorphic rearrangements correspond to operations that leave unaltered the nesting organization of the schema, including label reordering, adding and removing within a same nesting level.

Label reordering. Label reordering consists of modifying the order of labels to work around the mismatch between the orderings of source columns and atomic labels of the target schema. For example, without moving of label `Price` to the top of labels `Quantity` and `ProdName` in Figure 2.1(b), the mapping `OrderDetails`=A3:C5 would not have been possible.

Ignoring labels. If a table in the spreadsheet contains fewer columns than the number of atomic labels of the target schema, the structural mapping will not work properly since the range will not match the number of atomic labels. To this end, users can ignore a label through a context menu. The resulting display leaves the label in place but shows it in gray and with a red cross icon. This allows easy reversal of the operation. Suppose that two columns B2:B50 and C2:C50 in the spreadsheet in Figure 3.1 are merged with delimiter whitespace into column B containing names of customers (e.g., “Ford Prefect”). While the spreadsheet has 4 columns, the target schema consists of 5 atomic labels. The user can, for instance, ignore label `LastName` to make the structural mapping `Orders` =A2:D50 work properly. Values of labels `Id`, `ProdName`, and `Quantity` are correctly obtained from columns A2:A50, C2:C50, and D2:D50, respectively. To correct the mapping associated with label `FirstName` whose values are currently customer names, mapping `FirstName` =left(B2:B50, search(' ', B2:B50)) is used to refine. Then, label `LastName` is recovered and associated with mapping `LastName` =right(B2:B50, len(B2:B50)-search(' ', B2:B50)).

Adding labels. Conversely, if a table in the spreadsheet contains more columns than the number of atomic labels of the target schema, users can add new labels and give them any name provided it is not already used in the schema. No mapping formula evaluation is shown for these labels. Suppose that labels `FirstName` and `LastName` of the target schema in Figure 3.1 are merged into label `Name`. While the spreadsheet contains 5 columns, the target schema consists of 4 atomic labels. To make the structural mapping `Orders` =A2:E50 work properly, the user can add a new label located right below label `Name`. The mapping associated with `Name` must then be refined (it is incorrectly mapped with B2:B50): `Name` =concatenate(B2:B50, “ ”, C2:C50).

Anisomorphic view rearrangements

This category of view rearrangements corresponds to the case where labels are nested in the target schema in a way that does not match the nesting used in the spreadsheet. For example, the spreadsheet represented in Figure 4.1(a) is not isomorphic to the target `Quotation Request` schema (Figure 3.1(b)), although it uses exactly the same data. The difference is that spreadsheet in Figure 4.1(a) groups data per product while the target schema groups them per order. TranSheet allows users to rearrange the schema in a way that matches this spreadsheet via drag&drop manipulation of labels. In the background, TranSheet produces tgds [10, 20] to describe the mapping where the source schema is the new view and the target schema is the original schema. The user specifies mappings on this view and then the generated document is translated into a new document conforming to the original schema by executing tgds. For example, TranSheet generates the following tgd to describe the mapping between the restructured schema in Figure 4.1(b) and the original one in Figure 3.1(b):

$\forall$d $\in$ QuoteRequest.OrderDetails, o $\in$ d.Orders $\rightarrow$ $\exists$o' $\in$ QuoteRequest.Orders | o'.Order.Id = o.Order.Id, o'.Order.ShipTo.FirstName = o.Order.ShipTo.FirstName, o'.Order.ShipTo.LastName = o.Order.ShipTo.LastName, [$\forall$d2 $\in$ OrderDetails, o $\in$ d2.Orders $\rightarrow$ $\exists$d' $\in$ o'.Order.OrderDetails | d'.OrderLine.ProdName = d2.ProdName, d'.OrderLine.Quantity = o.Order.Quantity]

4 Generalizing mapping formulas

In this section, we focus on how a mapping formula can be generalized to transform multiple spreadsheet documents containing different data but organized according to the same template. We first present how native functions, available in common spreadsheet environments, can be used for generalizing mapping formulas. We then present new extensions to the native spreadsheet formula language allowing a wider class of generalizations.

4.1 Generalization using native spreadsheet formula functions

A mapping formula is not specific to a spreadsheet instance. It is applicable to a class of spreadsheet instances where cells at same locations have the same types. For instance, changing the value of cell E2 in Figure 3.1(a) from 150 to, say, 200 has no impact on the exportation mapping formula of Figure 3.1(b) (i.e, this formula could be used to export both spreadsheet instances). However, the organization of data that can be exported using the mapping in Figure 3.1(b) is very constrained.

For instance, this mapping can be used only for documents with exactly 49 items. This is due to the formula `Orders`=A2:E50 which indicates a fixed size range. It is desirable that adding or removing a row in the table does not invalidate the mapping (i.e, the mapping should be generic enough to export tables of any size).

Spreadsheet environments already provide native functions that are useful for generalizing mappings. For instance, using MS Excel formula language, the mapping of Figure 3.1(a) can be expressed with `Orders`=OFFSET(A1, 0, 0, COUNTA(A:A), 5).

This formula returns a range starting at cell A1 and spanning 5 columns. This range is dynamic since the number of rows is computed using COUNTA(A:A), which returns the number of non-empty cells in column A. Users familiar with such notation can readily apply this knowledge in specifying exportation formulas. However, there are situations where built-in spreadsheet functions are not sufficient for expressing required mappings. We discuss these situations in the next section and provide extensions to the mapping formula language to overcome them.

4.2 New notations for generalizing mappings

The organization of data in the spreadsheet shown in Figure 4.1(a) could be described, as follows:

"A list of product names each followed by its corresponding orders."

In order to capture this intuitive description of the spreadsheet content, the mapping formula language has to provide:

- A means to express the spatial location of entities by reference to each other. In this example, orders are located one row below product names.
- A means to control iterations over collections of cells. In this example, there are two iterations. First, the list of products needs to be enumerated and then, for each product, the list of its corresponding orders also needs to be enumerated. An additional difficulty is that product names are not located in consecutive rows, which makes traditional range expressions unusable.

We propose extensions to the formula language of existing spreadsheet environments in order to address the above requirements. Figure 4.1(b) illustrates how these extensions can be used to export the spreadsheet in Figure 4.1(a):

$$\begin{aligned}\texttt{ProdName} &= \mathrm{A1{:}A}\langle \mathbf{next{=}bottom}(\mathrm{Orders}) + 2\rangle \\ \texttt{Orders} &= \mathrm{A}\langle \mathbf{bottom}(\mathrm{ProdName}) + 1\rangle\mathrm{{:}D}\langle \mathbf{value}{=}\mathit{empty}\rangle\end{aligned}$$

These formulas express the following mappings. The first formula states that the first product name can be found in cell A1 and the subsequent product names are located one row after the end of the list of orders. The second formula states that orders are located in ranges spanning from column A to D and, in terms of rows, spanning from after the row containing a product name until the next empty row. The recursion stops when an empty value of product name is found.

Specifying relative location of spreadsheet data

A natural way to describe the content in a spreadsheet is by indicating the relative location of data. For instance, one may describe prices as being located in the column to the right of that containing quantities.

TranSheet allows users to refer, when specifying a mapping for a given label, to the "location" of other labels of the schema. By location of a label l, we mean the coordinates on the spreadsheet of a cell or a range of cells from which the value(s) of l is(are) derived. As detailed in previous sections, values are obtained from the spreadsheet by specifying a coordinate in the mapping formula (e.g. `Quantity`=A3), or by formula interpretation (see Section 3.2).

Given a label $l = (x_1, y_1) : (x_2, y_2)$, its value(s) can be obtained through four functions: **top**(l)=y_1, **left**(l)=x_1, **bottom**(l)=y_2, **right**(l)=x_2. For example, considering the mapping $l = A2 : C5$, we have top(l)=2, left(l)=1, right(l)=3 and bottom(l)=5.

Dynamic Length Ranges

We showed in Section 4.1 that it is possible to dynamically define the length of range expression using native spreadsheet functions. In this section, we extend the range notation to allow expressing ranges of dynamic lengths. Two extensions are proposed:

Dynamic range boundaries Users can refer to the location of other labels for indicating the boundaries of ranges. For instance, the range
A⟨bottom(Order)-1⟩:⟨right(Order), bottom(Order)+5⟩

	A	B	C	D
1	**Towel**			
2	0042	*Ford*	*Prefect*	2
3	0525	*Arthur*	*Dent*	1
4				
5	**Beer**			
6	0042	*Ford*	*Prefect*	150
7	0007	*Zaphod*	*Beeble*	300
8				
9	**Fish**			
10	0525	*Arthur*	*Dent*	1
11		...		

(a) Source spreadsheet with data grouped per product name

```
QuoteRequest
 Account
  Login =“MyLogin” {MyLogin}
  Password =“MyPass” {MyPass}
 OrderDetails [ ] {8 items}
  ProdName=A1:A⟨next=bottom(Orders)+2)⟩{Towel,...}
  Orders[ ]=A⟨bottom(ProdName)+1⟩:D⟨value=empty⟩
   Order
    Id {{0042, 0525, ...}, ...}
    ShipTo
     FirstName {{Ford, Arthur, ...}, ...}
     LastName {{Prefect, Dent, ...}, ...}
    Quantity {{2, 1, ...}, ...}
```

(b) Schema view and mapping specification

Figure 4.1: A generalized mapping specification for exporting datasets of varying sizes

corresponds to fixed number of rows (7 rows) spanning from column A to the left-most column of the range associated to label `Order`.

Conditional range boundaries. Often, the presentation style of data is used to identify data semantics in a spreadsheet. Users may rely on a visual style, an empty row or a border (i.e., a line surrounding a group of cell) to isolate data from each other. For example, an empty row is used in Figure 4.1(a) to isolate the list of orders of a product from the next product. TranSheet allows users to indicate boundaries of a range through conditions on the visual styles used to isolate data. For example, to specify that a range ends at an empty row, one may use A1:A⟨**value**=empty⟩.

In Figure 4.1(b), both above extensions are used in the formula `Orders`=A⟨bottom(ProdName)+1⟩:D⟨value=empty⟩. This formula uses a range expression such that:

- The range left-most and right-most columns are fixed (i.e., A and D respectively);
- The top-most row coordinate is given by “bottom(ProdName)+1”, meaning that the range starts one row after the product name;
- The bottom-most row is defined as the last non-empty row through the condition “value=empty”;

Mapping of non-adjacent collections of cells

Range expressions are convenient for enumerating collections of cells. The previous examples illustrated that through mapping collection of values to their corresponding labels. However, the product names that appear on the spreadsheet illustrated in Figure 4.1(a) cannot be enumerated easily using a range expression because the various product names are not stored in adjacent cells.

Existing spreadsheet environments provide a notation for ranges of non-contiguous cells which consists of enumerating each cell of the range (see Section 2.1). Using this notation, product names in Figure 4.1(a) could be mapped to the `ProdName` label using `ProdName` = A1,A5,A9.

However, the above notation is not convenient since it imposes to manually enumerate the location of each product name in the spreadsheet. This may be acceptable for small datasets but does not scale to larger ones. Another problem is that the exact locations of cells containing product names may vary from a spreadsheet instance to another. For instance, inserting a row after row 4 in Figure 4.1(a) to add a new towel order would render the above mapping invalid.

To alleviate this problem, we introduce a range notation that allows specifying the location of a first cell of a range and of subsequent cells through the keyword "next". Intuitively, the main reason why a collection of cells is not contiguous is because there are cells containing different information in between. Coming back to the example of Figure 4.1(a), the various product names are not located in contiguous cells because there are order details in between them. Considering a given product name cell (e.g., cell A1), the "next" product name is located after its corresponding list of orders (in this case, cell A5). This is specified using the mapping `ProdName`=A1:A⟨**next=bottom**(`Orders`)+2)⟩.

The above formula uses a reference to the label `Orders` to denote the location of each subsequent product names (i.e., two rows after the last row (bottom) of orders). The keyword "next" can be used for any (or both) of the two dimensions of a range. In its absence, rows and columns of a range are enumerated one by one. By default, iterations specified via **next** stop when an empty cell is found. Stopping criteria can be modified using conditions on other visual styles, such as font and border styles.

5 Mapping formula interpretation

We first illustrate how tgds are used to formally describe mappings of TranSheet and focus on the new functions that we introduce in tgd expressions. We then present the novelty in query generation from tgds.

5.1 TGD Generation

To employ a structural mapping (Section 3.2), TranSheet makes two assumptions which can be overridden. One of them is the ordering of source columns is identical to the ordering of target atomic labels. That is, when traversing the spreadsheet and the target schema from left to right, the first and the second columns correspond to the first and the second atomic labels, respectively, and so on. Consider mapping `Orders`=A2:E50 in Figure 3.1, source columns A2:A50, B2:B50, C2:C50, D2:D50, E2:E50 correspond to target atomic labels `Id`, `FirstName`, `LastName`, `ProdName`, `Quantity`, respectively. Let us represent these columns by the relation `Orders(Id,FirstName,LastName,ProdName,Quantity)`. Note that the names of the relation and the attributes can be arbitrary in implementation, but for the sake of readability we choose names that are identical to labels of the target schema. Given the above correspondences, using the existing mapping generation algorithm [10, 20], the following tgd (using the syntax of Clip [20]) is emitted to describe mapping `Orders`=A2:E50:

```
∀o ∈ Source.Orders → ∃o' ∈ QuoteRequest.Orders, d' ∈ o'.OrderDetails | o'.Order.Id = o.Id,
o'.Order.ShipTo.FirstName = o.FirstName, o'.Order.ShipTo.LastName = o.LastName,
d'.OrderLine.ProdName = o.ProdName, d'.OrderLine.Quantity = o.Quantity
```

Similarly, all mappings in Sections 3 and 4 can be represented using tgds.

Filtering. The filtering mapping example presented in Section 3.2 can be described by the following tgd:

```
∀o ∈ Source.Orders | (o.ProdName = ''Towel'') && (o.Quantity > 1) → ∃o' ∈
QuoteRequest.Orders, d' ∈ o.OrderDetails | o'.Order.Id = o.Id, o'.Order.ShipTo.FirstName =
o.FirstName, o'.Order.ShipTo.LastName = o.LastName, d'.OrderLine.ProdName = o.ProdName,
d'.OrderLine.Quantity = o.Quantity
```

Grouping with aggregation. Tgd for grouping with aggregation the mapping example in Section 3.2 is as follows:

```
∃groupby, count, max(∀o ∈ Source.Orders → ∃o' ∈ Target.Orders | o' = groupby(⊥,
o.OrderId, o.FirstName, o.LastName) o'.Order.Id = o.Id, o'.Order.FirstName = o.FirstName,
o'.Order.LastName = o.LastName, o'.Order.ProdName = count(o.ProdName), o'.Order.Quantity =
max(o.Quantity))
```

Join. Tgd corresponding to the join mapping example in Section 3.2 is:

```
 ∀o ∈ Source.Orders, c ∈ Source.Customers |o.Id = c.Id → ∃o' ∈ QuoteRequest.Orders, d' ∈
o.OrderDetails | o'.Order.Id = c.Id, o'.Order.ShipTo.FirstName = c'.FirstName,
o'.Order.ShipTo.LastName = c'.LastName, d'.OrderLine.ProdName = o.ProdName,
d'.OrderLine.Quantity = o.Quantity
```

Examples in Figures 4.1 and 2.1. In the mapping depicted in Figure 4.1, the mapping is specified at the atomic label `ProdName`, rather than at its corresponding $SetOf$ label
`OrderDetails`, which offers an increased flexibility compared with mappings expressed at the level of $SetOf$ labels. Regarding the source spreadsheet in Figure 4.1(a), for each product in the list of products, the product appears only once and the rest of its corresponding orders is "nested" within it. For example in Figure 4.1(a), there is one product Towel with two corresponding orders (with identifiers 0042 and 0525) located under it. This is also a natural way to organize spreadsheet data besides the tabular representation [15]. It is different from the spreadsheet in Figure 3.1(a) where each product explicitly appears in each row of the table.

In this example, TranSheet flattens each product with its corresponding order information in the source spreadsheet into a relation where attributes are identical to atomic labels of the target schema: `Orders`(`ProdName`, `Id`, `FirstName`, `LastName`, `Quantity`). Values of the attributes are computed based on the mapping formulas associated with labels of the target schema. Regarding the mapping in Figure 4.1 (suppose there are three products), we have:

`ProdName` =A1,A5,A9; `Orders` =A2:D3, A6:D7, A10:D10. Values of attributes `Id`, `FirstName`, `LastName`, `Quantity` are obtained from the mapping formulas associated with `Orders` by formula inheritance. For example, product Towel (i.e., `ProdName` =A1) is combined with 2 tuples of order identifier, first name, last name, and quantity, namely {0042, Ford, Prefect, 2} and {0525, Arthur, Dent, 1}, to form two new rows. It is similar for other two products Beer and Babel Fish. Finally, the following relational view is created:

	A	B	C	D	E	F
1	*ProdName*	*Id*	*FirstName*	*LastName*	*Quantity*	
2	Towel	0042	Ford	Prefect	2	
3	Towel	0525	Arthur	Dent	1	
4	Beer	0042	Ford	Prefect	150	
5	Beer	0007	Zaphod	Beeblebrox	300	
6	Babel Fish	0525	Arthur	Dent	1	

The tgd is then generated to describe the mapping between the relation and the target schema as follows:

```
∀o ∈ Source.Orders → ∃o' ∈ QuoteRequest.OrderDetails, d' ∈ o.Orders | o'.ProdName =
o.ProdName, d'.Order.Id = o.Id, d'.Order.ShipTo.FirstName = o.FirstName,
d'.Order.ShipTo.LastName = o.LastName, d'.Order.Quantity = o.Quantity
```

With respect to the mapping in Figure 2.1, the source spreadsheet (Figure 2.1(b)) is organized in a similar way where order items are "nested" within each customer detail. Note that the difference is that labels `Quantity`, `ProdName`, and `Price` are obtained values from range formulas associated directly with them, rather than inheriting from the *SetOf* label `OrderDetails`.

In the following, we focus on describing the mappings containing the new functions that we introduce to tgd expressions. To support sorting, we introduce the new function *sort(sorting-context, sorting-attribute1, sorting-order1,...)* where *sorting-context* is the scope of sorting; *sorting-attribute1* and *sorting-order1* are sorting attribute and its corresponding sorting order (with value "ASC" or "DESC"), respectively. For example, the tgd for the mapping example in Section 3.2 is:

```
∃sort(∀o ∈ Source.Orders → ∃o' ∈ QuoteRequest.Orders, d' ∈ o'.OrderDetails | o' = sort(⊥,
o.ProdName, ASC, o.Quantity, DESC), o'.Order.Id = o.Id, o'.Order.ShipTo.FirstName =
o.FirstName,...)
```

To support branching, we introduce the function *if(condition, value-if-true, value-if-false)* where *condition* is the preset condition; *value-if-true* is the value assigned to the function if *condition* is true; *value-if-false* is the value assigned to the function if *condition* is false. For example, the tgd corresponding to the mapping example in Section 3.2 is:

```
∃if(∀o ∈ Source.Orders → ∃o' ∈ QuoteRequest.Orders, d' ∈ o'.OrderDetails | o'.Order.Id =
o.Id,..., d'.OrderLine.Description = if(o.Quantity>20, 'large', 'small'))
```

To support the set operators union, intersect, and minus, we introduce the functions *union(variable1, variable2,...)*, *intersect(variable1, variable2,...)*, and *minus(variable1, variable2,...)* where *variable1*, *variable2*, and so on are set source variables. The tgd for the mapping example in Section 3.2 is:

```
∃union(∀o1 ∈ Source.Orders1, o2 ∈ Source.Orders2 → ∃o' ∈ QuoteRequest.Orders | o' =
union(o1, o2))
```

To support various patterns of value mapping, we introduce a collection of Excel-like functions in tgd expressions, such as *left*, *right*, *search*, *len*, and *concatenate*. For example, the tgd for the mapping example in Section 3.3 is:

```
∃left,search,right,len(∀o ∈ Source.Orders → ∃o' ∈ QuoteRequest.Orders, d' ∈
o'.OrderDetails | o'.Order.Id = o.Id, o'.Order.ShipTo.FirstName = left(o.Name, search('
',o.Name)), o'.Order.ShipTo.LastName = right(o.Name,len(o.Name)-search('
',o.Name)),d'.OrderLine.ProdName = o.ProdName,...)
```

5.2 Query Generation

Once the tgds have been generated, they are used to produce executable query XQuery for transformation as presented in [20]. In what follows, we only focus on the novelty involving XQuery generation for the new functions described above.

The *order by* clause of the XQuery FLWORs is used to generate query for function *sort* in a tgd expression. The *order by* clause consists of one or more ordering specifications, separated by commas. Each specification contains an ordering attribute and its related sorting order, corresponding to parameters *sorting-attribute1* and *sorting-order1* of function *sort*. The XQuery for the mapping example in Section 3.2 is:

```
<QuoteRequest>
{
  for $o in Source/Order
    let $p = $o/Price
    let $q = $o/Quantity
    order by $p ascending, $q descending
    return
    <Orders>
      <Order>
         <Id>{$o/Id/text()}</Id>
         ...
}</QuoteRequest>
```

The *if-then-else* construct of XQuery is used to generate query for function *if* in a tgd expression. While branch *if-then* corresponds to *value-if-true*, branch *then-else* corresponds to *value-of-false*. For example, the XQuery for the mapping example in Section 3.2 is:

```
<QuoteRequest>
{
  for $o in Source/Orders
  return
  ...
    if ($o.Quantity>20)
    then <Description>large</Description>
    else <Description>small</Description>
  ...
} </QuoteRequest>
```

XQuery provides the union, intersect, and except operators that are used by TranSheet to implement functions *union*, *intersect*, and *minus*, respectively, in tgd expressions. For instance, in the case of function *union(variable1, variable2)*, the set node corresponding to *variable1* is unioned with the set node corresponding to *variable2*. The XQuery for the mapping example in Section 3.2 is:

```
<QuoteRequest>
{
  let $orders = Source/Orders1 union Source/Orders2
  for $o in $orders
  return
  <Orders>
    <Order>
      <Id>{$o/Id/text()}</Id>
      ...
} </QuoteRequest>
```

For each function f of TranSheet appearing in tgds for value mappings, if there is a direct correspondence with a function f_X of XQuery, we use f_X in XQuery expressions. Otherwise, we create a new XQuery user-defined function whose name and parameters are identical to those of f. For example, while function *len* directly corresponds to function *string-length* of XQuery, function *left* has no direct correspondence. As a result, function *left* is defined as an XQuery function as follows:

```
declare function local:left($str as xs:string,$num_char as xs:integer)
                           as xs:string {
        return substring($str,1,$num_char); }
```

6 Graphical-based transformation utilities

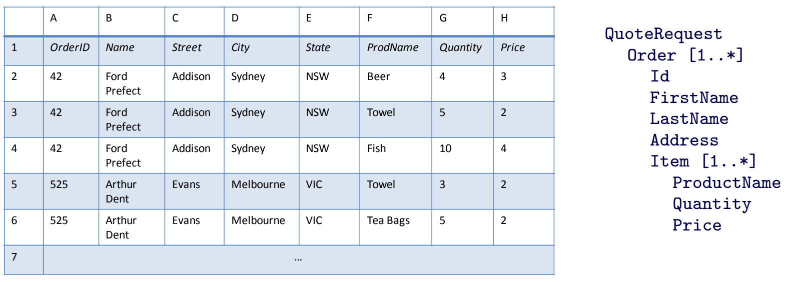
Some people use spreadsheets for nothing more than managing and printing a list of data items. Others know how to use very simple formula, such as $A11 = SUM(A1 : A10)$, but nothing more. As a result, the mapping language may be complex for them. Furthermore, even users who are already familiar with the language also sometimes wish to boost their productivity by not having to remember and type complicated syntax of the language. We argue that it is important to provide a technique to support specifying transformation graphically, rather than writing complex formulas from scratch. To this end, in this section, we provide graphical-based transformation utilities, including drag-and-drop operations and form-based transformation wizards (operators).

Similar to [21], for simple mappings like copying (e.g., mappings `Id` =A1, `ProdName` =C3:C5, `OrderDetails` =B3:C5 in Figure 2.1), the user can select a single cell or a (mono-dimensional or bi-dimensional) range and *drag-and-drop* it onto a target label. The corresponding mapping formula is then automatically generated for the label.

For example, to specify the mapping formula `OrderDetails` =B3:C5 in Figure 2.1, the user select the range B3:C5 using mouse and then drag it onto the label `OrderDetails`. After that, this mapping formula is automatically generated and displayed in the formula editor.

In what follows next, we present a set of visual transformation operators that complements the formula mapping language presented earlier. These operators solve two issues: (i) they enable users who do not have expertise in spreadsheet programming to specify transformation easily; (ii) they boost the productivity of users who are already experts of spreadsheet programming. They combine the power of graphical visualization and transformation patterns. Each operator characterizes a basic transformation pattern and is represented as a customizable form. The transformation from source to target is performed via composing multiple operators. A transformation operator is usually activated from the *contextual* menu associated with one target label and one or more columns of the source spreadsheet. It is contextual because only operators that are available to the current selected target label and source columns are shown up to the user.

Consider the following running example in Figure 6.1 for this section. Source spreadsheet containing order information grouped by order identifiers is shown in Figure 6.1(a). It needs to be mapped to the target schema shown in Figure 6.1(b).



	A	B	C	D	E	F	G	H
1	*OrderID*	*Name*	*Street*	*City*	*State*	*ProdName*	*Quantity*	*Price*
2	42	Ford Prefect	Addison	Sydney	NSW	Beer	4	3
3	42	Ford Prefect	Addison	Sydney	NSW	Towel	5	2
4	42	Ford Prefect	Addison	Sydney	NSW	Fish	10	4
5	525	Arthur Dent	Evans	Melbourne	VIC	Towel	3	2
6	525	Arthur Dent	Evans	Melbourne	VIC	Tea Bags	5	2
7	…							

```
QuoteRequest
  Order [1..*]
    Id
    FirstName
    LastName
    Address
    Item [1..*]
      ProductName
      Quantity
      Price
```

(a) (b)

Figure 6.1: (a) Source spreadsheet; (b) Target schema

6.1 Transformation Operator Definition

Given an operator, the corresponding form has various components that correspond to different parts of the operator. More specifically, while the left side contains components for the source, the right side is for the target components. A form is defined as a collection of *form-elements* (elements) laid out according to their purpose and relationships among them. A form-element is an object (i.e., form control) designed to translate a user's input into a basic fragment of the operator. For example, the drop-down list with selected column C2:C50 in Figure 6.2 indicates one element of the source.

The arrangement of elements in the form (possibly along with individual labels) indicates to the user what each element denotes and how it relates to other elements. The layout of these elements may involve organizing them into collections spatially within the form, and intuitively labelling each collection to show the purpose of the arrangement which is termed *form-groups* (groups). In other words, a form-group is simply a set of related elements and other possible groups organized in a labelled group. For example, group `Delimiter` in Figure 6.2 contains a set of delimiter elements represented by radio button controls.

In the following, we formally define transformation operators in a generic way so that it is possible to cover numerous transformation patterns.

Definition 6.1 *A transformation operator is represented as a tuple $O = (LP,RP)$ where LP is the left panel (source panel) and RP is the right panel (target panel). LP and RP contain the source and target components, respectively. Each panel P (either LP or RP) is characterized by an ordered set $\{g_1,...,g_n\}$ where:*

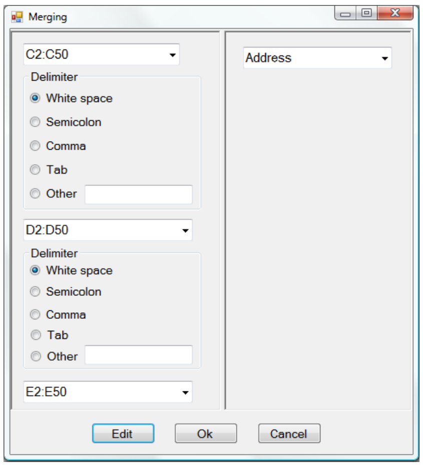


Figure 6.2: Merging Operator

- *g_i is a form-element or a form-group, $i \in \{1, ..., n\}$*
- *g_i is located above g_j in P if $i < j$*
- *$g_i(s)$ are arranged in the order expected by the transformation pattern that the operator represents.*

P itself is considered as the root (outermost) group.

Example 6.2 *While the source panel of the `Merging` operator depicted in Figure 6.2 consists of the set {`C2:C50`, `Delimiter`, `D2:D50`, `Delimiter`, `E2:E50`} where `C2:C50`, `D2:D50`, and `E2:E50` are three source columns and `Delimiter` is a group of delimiters, the target panel contains only one target element `Address`.*

6.2 Transformation Operator Customization

In this section, we present a set of available form customization operators offered by our system. We describe them in an abstract manner and show how they are realized.

Customization Operator Definition

In the following, we formally define form customization operators.

Form-element Insertion (α) This operator adds a new form-element to a panel (either source or target panel) or an existing group of the panel. Remember that the panel is considered as the root group. We can express this operator as $\alpha_{e,g}(P)$ where e is the form-element to be added, g is the group to which e is added, and P is a panel:

- $P' = \alpha_{e,g}(P) = \{g_1, ..., g_n, e\}$ if $g = P$.
- $P' = \alpha_{e,g}(P) = \{g_1, ..., g', ..., g_n\} | g' = \alpha_{e,g}(g)$ if $g \in P$.

As can be seen, the result of applying the operator to panel P is a new panel P'.

Form-element Deletion (β) This operator removes an existing form-element from a given panel or a group of the panel. We can define this operator as $\beta_{e,g}(P)$ where e is the form-element to be removed, g is the group from which e is removed, and P is a panel:

- P' $= \beta_{e,g}(P) = \{g_1, ..., g_{i-1}, g_{i+1}, ..., g_n\}$ where $e = g_i$, $i \in \{1, ..., n\}$ if $g = P$.
- P' $= \beta_{e,g}(P) = \{g_1, ..., g', ..., g_n\} | g' = \beta_{e,g}(g)$ if $g \in P$.

Form-element Move Up (χ) This operation moves up a form-element one position in a given panel or a group of the panel. This operator is expressed as $\chi_{e,g}(P)$ where e is an existing element of either panel P or group g of panel P:

- $P' = \chi_{e,g}(P) = \{g_1, ..., g_i, g_{i-1}, ..., g_n\}$ where $e = g_i$, $i \in \{2, ..., n\}$ if $g = P$.
- $P' = \chi_{e,g}(P) = \{g_1, ..., g', ..., g_n\} | g' = \chi_{e,g}(g)$ if $g \in P$.

Form-element Move Down (δ) This operation moves down a form-element one position in a given panel or a group of the panel. This operator is expressed as $\delta_{e,g}(P)$ where e is an existing element of either panel P or group g of panel P:

- $P' = \delta_{e,g}(P) = \{g_1, ..., g_{i+1}, g_i, ..., g_n\}$ where $e = g_i$, $i \in \{1, ..., n-1\}$ if $g = P$.
- $P' = \delta_{e,g}(P) = \{g_1, ..., g', ..., g_n\} | g' = \delta_{e,g}(g)$ if $g \in P$.

Form-group Insertion (ϵ) This operator inserts a form-group into a given panel. We can express this operator as $\epsilon_g(P)$ where g is a form-group to be inserted into panel P:

- $P' = \epsilon_g(P) = \{g_1, ..., g_n, g\}$

Form-group Deletion (η) A form-group can be removed from a panel using this operator. We can express this operator as $\eta_g(P)$ where g is a form-group of panel P:

- $P' = \eta_g(P) = \{g_1, ..., g_{i-1}, g_{i+1}, ..., g_n\}$ where $g = g_i$, $i \in \{1, .., n\}$

Form-group Move Up (γ) This operator is used to move up an existing form-group one position. It is defined as $\gamma_g(P)$ where g is a form-group of panel P:

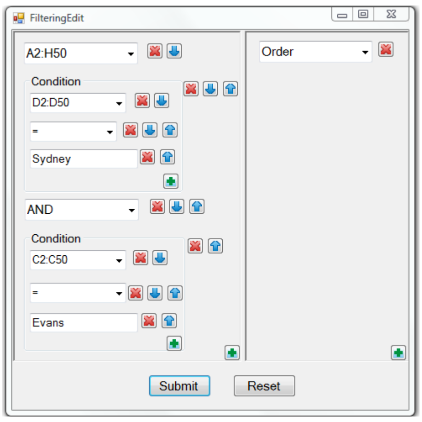
- $P' = \gamma_g(P) = \{g_1, ..., g_i, g_{i-1}, ..., g_n\}$ where $g = g_i$, $i \in \{2, .., n\}$

Form-group Move Down (λ) A form-group can be moved down one position in a given panel. It is defined as $\lambda_g(P)$ where g is a form-group of panel P:

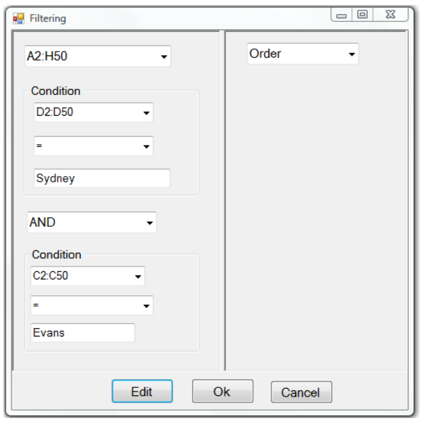
- $P' = \lambda_g(P) = \{g_1, ..., g_{i+1}, g_i, ..., g_n\}$ where $g = g_i$, $i \in \{1, .., n-1\}$

Customization Operator Generation

With any form as a starting point, the user can edit it using the form editor in multiple iterations until the desired form is obtained. The *editing mode* provides button-activated operations to modify the form. For instance, to enter editing mode of Figure 6.3(b), the user clicks on button `Edit` and the form editor is shown accordingly in Figure 6.3(a). To complete customization, the user clicks on button `Submit` (Figure 6.3(a)). To reset the form editor to the initial state, the user clicks on button `Reset`. In the following, we describe how each customization operator is realized.



(a)



(b)

Figure 6.3: (a) Editing Mode of Filtering Operator; (b) Filtering Operator

Form-element/Form-group Insertion For the sake of simplicity, form-element and form-group insertion are performed via the same insertion button marked "+" (Figure 6.3(a)). When this button is activated from the root group, the user selects either form-elements or form-groups from one of two form-panes: one for the form-elements and the other for form-groups. For example, when the user clicks on the insertion button in the source panel (i.e., the root group) of operator `Filter` in Figure 6.3(b), a new form is shown up which enables the user to select a suitable form-element (i.e., source elements and logical operators) or form-group (i.e., conditions). Note that form-elements and form-groups are displayed depending on the group in which the insertion button is activated.

Form-element/Form-group Deletion Form-elements or form-groups of a form that are irrelevant for current transformation can be removed from the form by clicking on remove button marked "X" (Figure 6.3(a)). In the case of form-group deletion, if the group is not empty, the user is asked whether to delete a non-empty group. For example, in editing mode of operator `Filter`, each form-element and each form-group are associated with a remove button. This conveniently allows the user to remove the form-element or the form-group.

Form-element/Form-group Move Up The user can move up a form-element or a form-group of a form one position by clicking on the move-up button marked "⇑" (Figure 6.3(a)). This operator is used to arrange form-elements/form-groups in the order expected by a transformation. In Figure 6.3(a), while all elements and groups except the top element A2:H50 of the source panel are associated with move-up buttons.

Form-element/Form-group Move Down An element or a group of a form can be moved down one position by clicking on the move-down button marked "$\Downarrow$" (Figure 6.3(a)). Similar to element/group move-up operator, this operator is used to arrange the elements/groups in the order expected by a transformation. In Figure 6.3(a), all elements and groups except the last group `Condition` of the source panel are associated with move-down buttons.

6.3 Design of Transformation Operators

In this section, we introduce specific transformation operators, define their semantics, and present their interface design based on the definition in and the customization mechanism in Section 6.2. Each operator will generate a corresponding formula after a completed customization.

Transformation Operators for Value Mappings

Merging This operator is designed to express value mappings of the form $t = \mathit{concatenate}(s_1, del_1, ..., del_{n-1}, s_n)$ where t is a target atomic element; s_i are source columns $i \in \{1, ..., n\}$; del_j are delimiters, $j \in \{1, ..., n-1\}$; function *concatenate* merges values of s_1,...,s_n into values of t where del_j are delimiters between s_j and s_{j+1}, $j \in \{1, ..., n-1\}$.

Figure 6.2 depicts the interface of Merging operator. Mapping formula `Address`=concatenate(C2:C50, " ", D2:D50, " ", E2:E50) of the running example (Figure 6.1) can be specified as follows. First, label `Address` and columns C, D, and E are selected; then the Merging operator is activated. A form is constructed with three source columns C2:C50, D2:D50, and E2:E50 in the source panel, and one target element `Address` in the target panel. The user turns on the editing mode and inserts two groups `Delimiter` between two pairs (C2:C50, D2:D50) and (D2:D50, E2:E50) using the customization operators presented in Section 6.2. Next, two delimiters white spaces " " of two newly added groups are chosen. Finally, the button `Submit` is clicked to complete customization.

Splitting Splitting operator is used to express value mappings of the form $\{t_1, del_1, ..., del_{n-1}, t_n\} = \mathit{split}(s)$ where t_i are target atomic labels; s is a source column; function *split* splits s into an array of values V in which $t_i = V[i]$ and del_j are delimiters between t_j and t_{j+1}, $i \in \{1, ..., n\}$ and $j \in \{1, ..., n-1\}$.

Figure 6.4 shows the interface of Splitting operator. In contrast to Merging operator, while the source panel contains one source column, the target panel contains multiple target attributes and groups `Delimiter`. Each group `Delimiter` consists of a list of pre-defined delimiters represented as radio buttons.

Two mappings of the running example `FirstName` =left(B2:B50, search(" ", B2:B50)) and `LastName` =right(B2:B50, len(B2:B50) - search(" ", B2:B50)) can be expressed as follows. The user selects the row B and labels `FirstName` and `LastName` and activates the Splitting operator. A new form is constructed with element B2:B50 in the source panel and two elements `FirstName` and `LastName` in the target panel. The user then customizes the form by adding

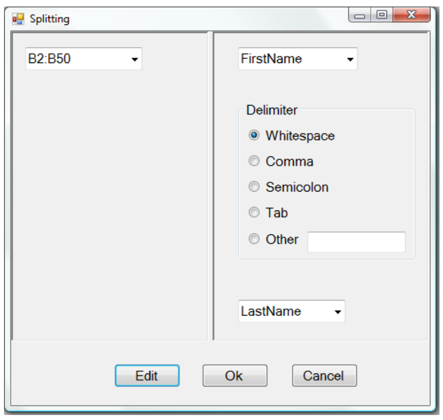


Figure 6.4: Splitting Operator

a new group `Delimiter` between `FirstName` and `LastName`, and ticks on radio button `Whitespace` of the group.

Constant Value Generation In some special cases, a target element do not correspond to any source spreadsheet content. The transformation operator for this pattern is defined as t =c where t is a target atomic label and c is a constant. The source panel contains a textbox for entering a constant and the target panel contains one atomic label.

Copying This is the simplest operator used for mappings t =s where t is a target element and s is one cell or a collection of cells. The user activates the Copying operator and values of s are copied into values of t. For example, mapping `ProductName` =F2:F50 of the running example can be performed by selecting column F and label `ProductName`, and activating the Copying operator.

Transformation Operators for Structural Mappings

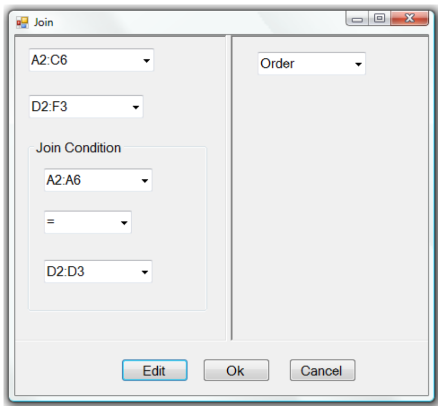
Filtering Filtering operator is designed to cover structural mappings of the form $t = s[filterexp]$ where t is a structural label; s is a two-dimensional range; *filterexp* is a filter expression associated with s.

Interface of Filtering operator is depicted in Figure 6.3(b). While the source panel contains one two-dimensional range, condition groups, and form-elements containing logical operators, the target panel contains one target structural label. Each condition group consists of an atomic element, a form-element storing comparator operators $\{=, <=, >=, !=, >, <\}$, and a textbox for entering values. The logical operator form-element containing {AND, OR} is used to combine condition groups.

For example, mapping `Order` = A2:H50[AND(D2:D50 = “Sydney”, C2:C50 = “Evans”)] of the running example can be specified as follows. The user first selects the range A2:H50 and label `Order` and activates the Filtering operator. A form is constructed with one element A2:H50 in the source panel, and the

	A	B	C	D	E	F
1	*OrderID*	*ProdName*	*Quantity*	*ID*	*FirstName*	*LastName*
2	42	Beer	180	42	Ford	Prefect
3	42	Towel	2	525	Arthur	Dent
4	42	Fish	1			
5	525	Towel	1			
6	525	Teabags	20			

Table 6.1: Two tables need to be joined



A2:C6

D2:F3

Join Condition

A2:A6

=

D2:D3

Order

Figure 6.5: Join Operator

label `Order` in the target panel. Then the user adds a condition group representing *D2:D50 = 'Sydney'*, logical operator element AND, and a condition group representing *C2:C50 ="Evans"*.

Join Join operator is used to express structural mappings of the form $t = join(s_1, s_2, joinexp?)$ where t is a structural target element, s_1, s_2 are two-dimensional ranges, and *joinexp* is an optional join condition associated with s_1 and s_2.

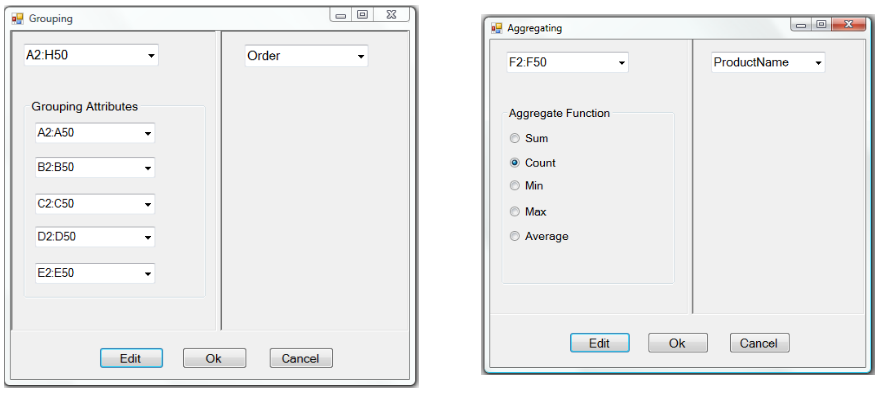
Note that *joinexp* is optional and if it is absent, the Cartesian product is calculated between s_1 and s_2. Figure 6.5 illustrates the interface of Join operator to join two tables in Table 6.1. While the source panel contains two two-dimensional ranges that involve in the join and one condition group for expressing join conditions, the target panel contains one structural target label. Each condition group contains two source columns and a comparator operator form-element between them.

Grouping with aggregation This kind of mapping allows users to group the source spreadsheet according to certain columns, and then aggregate functions can be applied on each group. In particular, the user needs to use two operators, namely Grouping and Aggregate.

The Grouping operator covers structural mappings of the form
$t = s[groupby(s_1, ..., s_n)]$ where t is a target structural label; s is a two-dimensional range; s_i are grouping columns of the source, $i \in \{1, ..., n\}$.

Figure 6.6(a) illustrates the interface of Grouping operator. While the source panel contains one two-dimensional range and one group consisting of a list of grouping columns, the target panel contains one structural target label. Mapping `Order` = A2:H50[groupby(A2:A50, B2:B50, C2:C50, D2:D50, E2:E50)] of the running example can be specified by selecting range A2:H50 and label `Order`, and activating the Grouping operator. A form is constructed with the range A2:H50 and the structural element `Order` for the source and target panels, respectively. The user adds a grouping attribute group to the source panel, and then inserts the source columns A2:A50, B2:B50, C2:C50, D2:D50, and E2:E50 to the group.

The Aggregate operator is used to express mappings of the form $t = aggrname(s)$ where t is a target atomic label; s is a source column; *aggrname* is one of functions in the set $\{sum,\ count,\ min,\ max,\ average\}$.



A2:H50

Grouping Attributes
A2:A50
B2:B50
C2:C50
D2:D50
E2:E50

Order

F2:F50

Aggregate Function
Sum
[✓ Count]
Min
Max
Average

ProductName

(a) (b)

Figure 6.6: Transformation Operators: (a) Grouping; (b) Aggregate

Figure 6.6(b) depicts the interface of Aggregating operator. Mapping `ProductName` =count(F2:F50) of the running example can be obtained by selecting row F2:F50 and label `ProductName`, and activating the Aggregate operator. Then the user needs to insert a aggregate function group, and choose function *count* in the group. Other mappings `Quantity` =sum(G2:G50) and `Price` = average(H2:H50) are performed similarly.

Sorting Mappings expressed by this operator is defined as
$t = s[sort(s_1, o_1, ..., s_n, o_n)]$ where t is a target structural label; s is a two-dimensional range; function *sort* is used to sort values of s according to source columns s_i in orders $o_i \in \{\text{ASC,DESC}\}$, $i \in \{1, ..., n\}$.

Figure 6.7 depicts the interface of Sorting operator for the mapping `Order` =A2:H50[sort(G2:G50, ASC, H2:H50, DESC)] of the running example. The source panel contains one two-dimensional range A2:H50 and a group consisting of sorting columns (G2:G50 and H2:H50) with their corresponding orders (ASC and DESC); the target panel contains one target structural label `Order`.

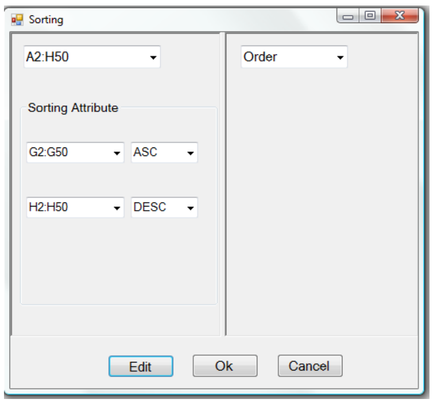


Figure 6.7: Sorting Operator

6.4 Transformation Operation Modification

Having introduced all operators, we describe how the user can modify or delete a specified operation. Suppose the user has performed n operations $\{O_1, O_2,...,O_n\}$ in sequence (i.e., O_i is performed earlier than O_j if $i < j$), and she wants to modify or delete the operation O_i, $1 \leq i \leq n$. In regard to our mapping interface, the user can interact with one transformation operation at a time and cannot see other specified transformation operations. On the contrary, in the case of relationship-based mapping tools [23], specified transformation operations can be seen via connecting lines and functions associated with those lines. Therefore, we provide a "History" list which contains all specified operations with corresponding mapping formulas in time order. By using this list, the user can modify or delete any operation she wants and resubmits it for evaluation. For example, in the case of the running example, we have the following list:

```
1. Copying (Id =A2:A50) Modify|Delete
2. Copying (ProductName =F2:F50) Modify|Delete
3. Copying (Quantity =G2:G50) Modify|Delete
4. Merging (Address=concatenate(C2:C50,' ',D2:D50,' ',E2:E50)) Modify|Delete
5. Copying (Price =H2:H50) Modify|Delete
6. Splitting (FirstName =left(B2:B50, search(' ',B2:B50))) &&
(LastName =right(B2:B50,len(B2:B50)-search(' ',B2:B50))) Modify|Delete
7. Filtering (Order=A2:H50[AND(D2:D50='Sydney',C2:C50='Evans')]) Modify|Delete
```

As can be observed from the above list, the first specified operation is copying with mapping formula `Id` =A2:A50; the last one is filtering with mapping formula `Order` =A2:H50[AND(D2:D50='Sydney', C2:C50='Evans')]. To modify or delete an operation in the list, the user clicks on the corresponding link under Modify and Delete, respectively.

7 Implementation and Experiments

7.1 Implementation

Architecture. TranSheet has been implemented as an Excel plug-in using C# 3.0 and Visual Studio 2008. Figure 7.1 depicts the architecture of TranSheet with the following main components: (i) GUI enables users to specify mapping via formulas. While spreadsheet data is imported using the built-in functionality of Excel, target schemas are imported using TranSheet functionality; (ii) Mapping generation engine takes input mapping formulas from GUI and generates corresponding tgds (Section 5.1); (iii) Query generation engine generates XQuery from input tgds (Section 5.2); (iv) Execution engine is responsible for executing input XQuery and then returning the transformation result to GUI for validation. TranSheet currently employs open-source execution engine Saxon [1].

User interface. The user interface of TranSheet is shown in Figure 7.2. While the left side corresponds to the source spreadsheet, the target schema is located in the Excel task pane on the right. To specify a mapping, the user selects a target label and enters a formula into the formula editor located in the task pane. Instant feedback for the mapping is then displayed adjacent to the target labels. Mapping formulas can be entered manually into the formula editor or automatically generated using GUI-based transformation ultilities as presented in Section 6.

When a mapping is specified for a label, it may violate some of the constraints attached to that label. For example, mapping `Quantity` =B3:B5 may conflict with type integer of label `Quantity` if range B3:B5 contains at least one non-integer value. In such case, TranSheet provides a warning on type mismatch next to label `Quantity`.

7.2 Experiments

In this section, we focus on evaluating two major benefits of TranSheet, namely the expressive power and mapping generalization. To evaluate the expressiveness, we compare TranSheet with Excel XML Mapping (MS Excel XML Mapping) [21] and IBM ManyEyes [26, 4] in the *end-user visualization* context, which maps spreadsheets to visualization types. To evaluate the effectiveness of mapping generalization, we consider a *medical data transfer* case study, which exports a collection of spreadsheets representing orthodontic patient records to the target schema of an office management application.

Expressiveness

Experimental setup. We selected 4 publically available real tabular data sets from site ManyEyes [4]. Each data set has column headers. The *swine flu* dataset is a table showing flu infection level for 193 countries with 6 columns and 194 rows. The *oil and gas* data set contains the number of wells drilled by counties in Pennsylvania from 2000 to 2010 with 12 columns and 38 rows. The *most expensive cities* data set indicates most expensive cities in the world from 2002 to 2009 with 9 columns and 144 rows. The *smart phone sales* data set

[1]http://saxon.sourceforge.net/

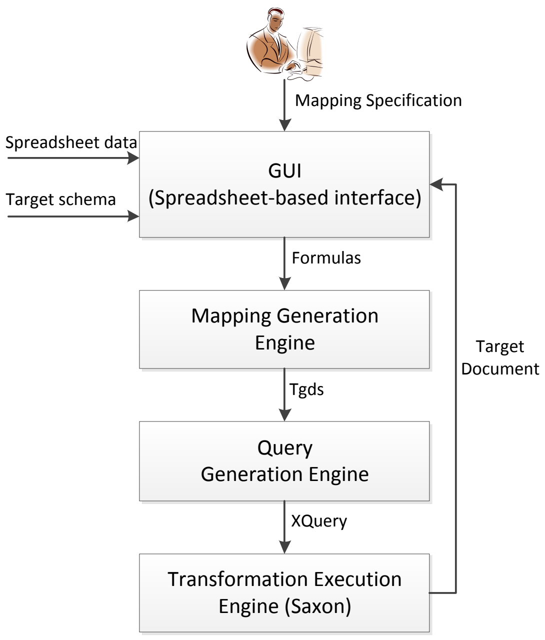
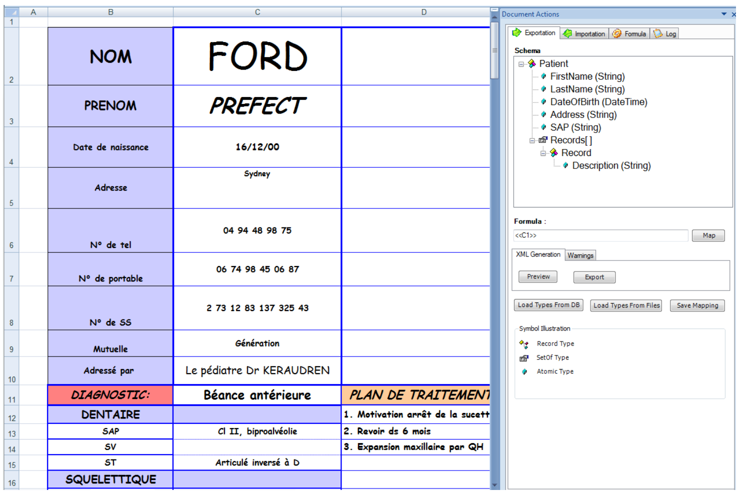


Figure 7.1: TranSheet architecture



	A	B	C	D
1				
2		NOM	FORD	
3		PRENOM	PREFECT	
4		Date de naissance	16/12/00	
5		Adresse	Sydney	
6		N° de tel	04 94 48 98 75	
7		N° de portable	06 74 98 45 06 87	
8		N° de SS	2 73 12 83 137 325 43	
9		Mutuelle	Génération	
10		Adressé par	Le pédiatre Dr KERAUDREN	
11		DIAGNOSTIC:	Béance antérieure	PLAN DE TRAITEMENT
12		DENTAIRE		1. Motivation arrêt de la sucett
13		SAP	Cl II, biproalvéolie	2. Revoir ds 6 mois
14		SV		3. Expansion maxillaire par QH
15		ST	Articulé inversé à D	
16		SQUELETTIQUE		

Schema
- Patient
  - FirstName (String)
  - LastName (String)
  - DateOfBirth (DateTime)
  - Address (String)
  - SAP (String)
  - Records[ ]
    - Record
      - Description (String)

Formula : <<C1>>

Symbol Illustration
- Record Type
- SetOf Type
- Atomic Type

Figure 7.2: TranSheet user interface

contains the number of smart phone units sold in millions by operating systems 2007-2011 (2010 and 2011 are projected numbers) with 3 columns and 31 rows.

Our test schemas are four structurally different visualization types *Pie Chart*, *Scatter Plot*, *Bar Chart*, and *World Map* of ManyEyes. Schemas corresponding to some visualization types are shown in Figure 1.1. All data sets and schemas can be found on our web page [2].

Eight popular mapping scenarios commonly used by the ManyEyes community for visualization [4] are considered in this experiment and are summarized in Table 7.1. Each row of the table indicates name of a mapping scenario, data set and visualization type to be used, and the mapping scenario description.

Methodology. Prior to implementing mapping scenarios, we familiarized ourselves with all functionalities offered by MS Excel XML Mapping and ManyEyes. For each mapping scenario in Table 7.1, if it can be implemented using a system, we record manipulation operations on the source data set, if any. These operations include column deletion (CD), column insertion (CI), row deletion (RD), row insertion (RI), data set sorting (DS), data set filtering (DF), and cell value changing (VC).

Observations. In a nutshell, TranSheet can implement all mapping scenarios using mapping formulas without modifying data sets. On the other hand, both MS Excel XML Mapping and ManyEyes need multiple manipulations on data sets to accomplish mapping scenarios (summarized in Table 7.2). MS Excel XML Mapping generally requires fewer manipulations than ManyEyes and multiple manipulations of MS Excel XML Mapping can be done using graphical wizards. This is because Excel provides many advanced features to support data manipulation. However, MS Excel XML Mapping cannot implement mapping scenario nesting+sorting. While MS Excel XML Mapping supports only one nesting level in the target schema, a bar chart consists of two nesting levels.

Copying is the only mapping scenario for which MS Excel XML Mapping and ManyEyes require no source modificaion. MS Excel XML Mapping supports copying by dragging target attributes onto columns containing county name and the number of drilled wells in 2005. In the case of ManyEyes, while the text target attribute is mapped with the column containing county name, the column containing the number of drilled wells in 2005 is selected from a list from 11 candidates (years 2000-2010) to map with the numeric target attribute. TranSheet supports copying via either range formulas or drag-and-drop operations.

Unlike copying, MS Excel XML Mapping and ManyEyes involve many manipulations to perform derivation, splitting, and merging. The two systems share the same number of manipulation operations in implementing these mapping scenarios. To implement derivation, values of the infection rate column must be divided by 10. A new column is inserted and its values are changed in the case of splitting or merging to store splitting/merging values. Note that in order to implement splitting, unwanted rows must be deleted first since only the top 20 rows are considered. Although Excel offers range notation, MS Excel XML Mapping selects by default the entire column, even if a specific range of the column is selected. TranSheet supports derivation, merging, and splitting via applying functions on range formulas. For instance, the function *sum(column1, column2,...)* is used to merge and calculate the total number of drilled wells in

[2]http://cgi.cse.unsw.edu.au/∼vthung/

Mapping Scenario	**Data Set**	**Visualization Type**	**Description**
Copying	Oil and gas	Pie Chart	Visualize the number of drilled wells in Pennsylvania counties in year 2005.
Derivation	Swine flu	Scatter Plot	Use a scatter plot find correlation between the number of confirmed cases and the infection rate in each country. The infection rate is visualized per one hundred thousand, instead of per million in the data set.
Merging	Oil and gas	Pie Chart	Visualize the total number of drilled wells in Pennsylvania counties in ten years from 2000 to 2009.
Splitting	Most expensive cities	World Map	Put the 20 most expensive cities in 2009 on the world map. Split the city information in the data set to get the country name for visualization.
Sorting	Oil and gas	Pie Chart	Visualize the number of drilled wells in Pennsylvania counties in descending order in year 2010.
Filtering	Swine flu	Scatter Plot	Use a scatter plot find correlation between the number of confirmed cases and the infection rate in each country, but select only countries whose confirmed cases are greater than 20000.
Grouping with aggregation	Smart phone sales	Pie Chart	Group the data set by year, average smart phone units in each year, and visualize each year with its corresponding average sales amount.
Nesting+Sorting	Swine flu	Bar Chart	Visualize a bar chart of the confirmed cases for 20 countries with the largest number of confirmed cases.

Table 7.1: Mapping Scenarios

Mapping Scenario/Tool	EXM	ManyEyes
	Source Manipulation	Source Manipulation
Copying	No modification	No modification
Derivation	193VC	193VC
Merging	1CI+37VC	1CI+37VC
Splitting	123RD+1CI+20VC	123RD+1CI+20VC
Sorting	1DS	37RD+37RI
Filtering	1DF	189RD
Grouping with aggregation	2CI+5RI+10VC	2CI+5RI+10VC+ 3CD
Nesting+Sorting	Not supported	20RI+193RD

Table 7.2: Source manipulation operations of EXM and ManyEyes in implementing mapping scenarios

ten years and to map with the numeric target attribute of pie chart.

MS Excel XML Mapping requires fewer manipulations than ManyEyes in implementing sorting and filtering since Excel provides corresponding graphical wizards for users to implement these mapping scenarios. The user needs to perform these manually in the case of ManyEyes. Although ManyEyes offers sorting functionality, it works only for text, not numbers. To sort the number of drilled wells in 2010 in descending order, for instance, the user must manually select, cut and paste all required rows individually. To filter countries whose confirmed cases are greater than 20000, rows containing confirmed cases equal or less than 20000 are deleted from the data set. TranSheet supports filtering and sorting by performing structural mappings via formulas or wizards. For instance, the structural mapping `Dots` =B3:G195[E2:E195>20000] is used for filtering.

To implement grouping with aggregation using MS Excel XML Mapping, a new table is created with 2 columns and 5 rows, in which each row contains a year and its average sales. To perform grouping, ManyEyes selects by default the text column containing operat- ing system names as a grouping attribute and calculates totals for two candidate columns, year and sales. Moreover, it supports only one grouping attribute and one aggregate function to compute total. As a result, this mapping scenario cannot be implemented using the default grouping of ManyEyes. Instead, like MS Excel XML Mapping, the user must create a new table and then deletes three old columns to avoid default grouping. TranSheet supports this mapping scenario by grouping the data set according to column year and then using the aggregate function *average*.

The nesting+sorting mapping scenario is a combination of two transformation patterns: sorting and nesting. In the case of ManyEyes, the user must first sort the data set in descending order of confirmed cases and then select top 20 rows. Both are done manually as described above. TranSheet supports this mapping scenario by selecting 20 desired rows individually (i.e., row with the maximum confirmed case is selected first and so on) via non-contiguous ranges. This may be, however, tedious when the data set is large. To address this issue, a new function to select a subset of tuples of the data set (e.g, *top(i,j)* where *i* is the position of the first selected tuple and j is the number of selected tuples) can be developed to use with function *sort*.

Mapping generalization

Experimental setup. The *medical* (orthodontic) dataset corresponds to more than 700 spreadsheet documents that were created and used by a French orthodontist. Each document contains the personal information and medical records of a patient. For each new patient, the orthodontist uses a new empty spreadsheet template and manually fills the necessary cells with the patient data. While following the same overall template, the resultant spreadsheets are slightly different from each other. For instance, both the list of treatments and the table for containing consultation times and charges vary in size from patient to patient. An extract of a document and the target schema is shown in Figure 7.2, in which the personal information is presented as single cells adjacent to their labels on the top, while the list of treatments is located at the bottom in column D. For confidentiality reasons, the source of data is not provided.

Methodology. We consider only MS Excel XML Mapping because ManyEyes is limited to predefined schemas for visualization. With respect to MS Excel XML Mapping, a mapping is bound to a specific document and, strictly speaking, exportation of multiple instances is not supported. However, this aspect can be ignored since we merely want to uncover the potential problems posed by the column-based approach based on drag-and-drop operations for mapping specifications on multiple structurally similar spreadsheets.

Observations. We encountered two types of problem when using MS Excel XML Mapping for the exportation of the medical dataset. The first is related to the exportation of lists and tables of varying size. For instance, a list of treatments is represented by a series of cells in a column and delimited only through visual clues (column D in Figure 7.2). To handle lists and tables of varying size, EXM insists that columns of lists or tables are first transformed into data lists. However, doing so clutters the spreadsheet by inserting header rows and it must be done *manually* for each spreadsheet instance. The second problem concerns the mapping of values that appear at varying coordinates depending on each spreadsheet. For instance, in Figure 7.2, the field `SAP` (which stands for Antero-Posterior Situation) is located in cell B13 and its value is in cell C13. This field may be located at another row in another spreadsheet, but always in column B or entirely missing. Therefore, to obtain the value for `SAP`, it is necessary to automatically lookup the coordinate of the cell containing this label and take the value of the cell located to its right. Looking up labels is usual in spreadsheet programming (e.g., Excel supports this feature through function *vlookup*). However, to exploit this feature for exportation, one would have to prepare a separate spreadsheet where each of the field values is assigned a definite cell whose value is obtained by a formula that uses the lookup function. Building such a spreadsheet is tedious and time-consuming. TranSheet addresses the first problem through mapping formulas for ranges of varying length (Section 4.2) and the second problem through the expression of relative locations (Section 4.2). For instance, we use the mapping formula `SAP` =C⟨**bottom**(B⟨**value**=”SAP”⟩)⟩ to obtain the value for the label `SAP`. This formula evaluates the value to the right of the cell in column B which contains the text “SAP”. The list of current treatments is retrieved by using the mapping formula: `Records` =D12:D⟨**value**=empty⟩

8 Related Work

Transformation Languages and Mapping Tools. A language for the description of spreadsheet content as a series of relational tables is proposed in [15]. Once defined, the schema can be used with either low-level transformation languages (e.g., XSLT/XQuery) or visual mapping tools (e.g., Clio [19, 10], Clip [20], and Altova MapForce [1]) to perform transformation. However, spreadsheet users must learn a new language to perform transformation and their existing programming experience is not leveraged. By contrast, TranSheet provides a familiar spreadsheet-like formula mapping language as well as GUI-based utilities to ease transformation specification.

Exportation of spreadsheet data. Several existing approaches support the exportation of spreadsheet data to structured formats [21, 8, 26, 11]. Regarding these approaches, data representation or structural mismatches between a source spreadsheet and target XML schema are addressed by modifying the spreadsheet document before exportation. Most of the transformation logic is embedded in modifications of the source spreadsheet. By contrast, TranSheet separates the transformation logic from source manipulations through the notion of mapping formulas. The benefits are increased expressiveness and preservation of the source document presentation. The reuse of a mapping to export multiple structurally similar spreadsheets is another benefit.

Schema Mapping and Data Exchange. In data exchange, given a source instance, there are many possible solutions for the target [9]. A *universal solution* has no more and no less data than required for the data exchange problem, and is therefore preferable. Each document generated by TranSheet corresponds to a canonical universal solution. We use the tgds to describe the semantics of TranSheet. With respect to existing approaches [10, 20], TranSheet introduces a collection of new functions in tgd expressions to cover numerous transformation patterns provided by other transformation languages and mapping tools [7]. TranSheet also extends a previous query generation algorithm [20] to generate XQuery for these new functions.

Spreadsheet-based data access and manipulation. The spreadsheet programming paradigm is leveraged in [16, 25] to simplify specification of SQL queries using formulas. Mashroom [27] used this paradigm to display the nested relational data and developed a set of mashup operators for that data model to build mashup applications. Our work is also based on the simplicity and effectiveness of the spreadsheet programming paradigm, but we focus on transforming spreadsheet data to XML.

Our previous work [14] focused on importation external data into spreadsheets and proposed a number of widgets to present such data. Some generic notations described in Section 4 are used to bind data to those widgets. Instead, in this paper those notations are differently employed to generalize mappings for transformation purpose. We also introduce new notations for specifying conditional range boundaries based on visual styles.

9 Conclusion and Future Work

In this paper, we have proposed an approach for transforming spreadsheet data to XML. The approach is based on a mapping language which reuses most of the familiar concepts of spreadsheet formulas and implements all common transformation patterns. It supports users through the immediate evaluation and preview of the transformation to help building incrementally the desired mapping, while keeping the source document unmodified. It also addresses the re-usability of the mapping for various spreadsheets through the concept of generalized mappings.

In the future, TranSheet will be augmented by a semantic matching module to (semi-)automatically suggest correspondences between spreadsheet cells and target labels. Currently, users must identify these correspondences manually. We will also conduct a user study to evaluate the usability of TranSheet.

Bibliography

[1] Atova mapforce. http://www.altova.com/mapforce.html.

[2] Excel web app. http://office.live.com.

[3] Google spreadsheet. http://docs.google.com.

[4] Manyeyes. http://many-eyes.com/.

[5] Zoho spreadsheet. http://http://sheet.zoho.com/.

[6] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VLHCC '04*.

[7] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *Proc. VLDB Endow.*, 1(1):230–244, 2008.

[8] Bob Brauer. Next evolution of data integration into microsoft excel. Technical report, StrikeIron Inc., 2006.

[9] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[10] Ariel Fuxman, Mauricio A. Hernandez, Howard Ho, Renee J. Miller, Paolo Papotti, and Lucian Popa. Nested mappings: schema mapping reloaded. In *VLDB '06*.

[11] Hector Gonzalez, Alon Halevy, Christian S. Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, and Warren Shen. Google fusion tables: data management, integration and collaboration in the cloud. In *SoCC '10*.

[12] Carlos Jensen, Heather Lonsdale, Eleanor Wynn, Jill Cao, Michael Slater, and Thomas G. Dietterich. The life and times of files and information: a study of desktop provenance. In *CHI'10*.

[13] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centered approach to functions in excel. In *Int. Conf. on Functional programming (ICFP'03)*, New York, USA, 2003.

[14] Woralak Kongdenfha, Boualem Benatallah, Julien Vayssière, Régis Saint-Paul, and Fabio Casati. Rapid development of spreadsheet-based web mashups. In *WWW*, 2009.

[15] L.V.S. Lakshmanan, S.N. Subramanian, N. Goyal, and R. Krishnamurthy. On querying spreadsheets. In *ICDE'98*, Los Alamitos, CA, USA, 1998.

[16] Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE '09*.

[17] Fabian Nunez. An extended spreadsheet paradigm for data visualisation systems, and its implementation, 2000. Master Thesis, University of Cape Town.

[18] J.D. Pemberton and A.J. Robson. Spreadsheets in business. *Industrial Management & Data Systems'00*, 2000.

[19] Lucian Popa, Yannis Velegrakis, Renee J. Miller, Mauricio A. Hernandez, and Ronald Fagin. Translating web data. In *VLDB'02*.

[20] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a visual language for explicit schema mappings. In *ICDE'08*.

[21] Frank Rice. Creating XML mappings in excel 2003. Technical report, Microsoft Corporation, 2005.

[22] George G. Robertson, Mary P. Czerwinski, and John E. Churchill. Visualization of mappings between schemas. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 431–439, New York, NY, USA, 2005. ACM Press.

[23] M. Roth, M. A. Hernandez, P. Coulthard, L. Yan, L. Popa, H. C.-T. Ho, and C. C. Salter. Xml mapping technology: making connections in an xml-centric world. *IBM Syst.J.'06*.

[24] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *VLHCC '05*.

[25] Jerzy Tyszkiewicz. Spreadsheet as a relational database engine. *SIGMOD'10*.

[26] F. B. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. Manyeyes: a site for visualization at internet scale. 2007.

[27] Guiling Wang, Shaohua Yang, and Yanbo Han. Mashroom: end-user mashup programming using nested tables. In *WWW'09*.

[28] Cong Yu and Lucian Popa. Constraint-based xml query rewriting for data integration. In *SIGMOD'04*, 2004.