

Spreadsheet-based complex data transformation

Author:

Vu, Hung Thanh

Publication Date:

2011

DOI:

<https://doi.org/10.26190/unsworks/23867>

License:

<https://creativecommons.org/licenses/by-nc-nd/3.0/au/>

Link to license to see what you are allowed to do with this resource.

Downloaded from <http://hdl.handle.net/1959.4/51313> in <https://unsworks.unsw.edu.au> on 2024-04-18

Spreadsheet-based complex data transformation

Hung Thanh Vu



Dissertation submitted in fulfilment
of the requirements for the degree of
Doctor of Philosophy

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

March 2011

Supervisor: Prof. Boualem Benatallah

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed



Date

12/09/2011

COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).

I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed



Date

12/09/2011

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed



Date

12/09/2011

Acknowledgements

I am very grateful to Professor Boualem for his exceptional unconditional support and limitless patience. He was the first person who taught me how to do research; how to write and present a complex research problem. He has always been there for me when I have any difficulties in research. He is one of the best supervisors I have ever worked with. Without his support, this thesis would never be completed.

My sincere thanks go to Dr Regis Saint-Paul for his fruitful collaborations and providing me invaluable research skills.

I also wish to express my gratitude to the members of the SOC group, who spent a lot of time discussing with me on the research issues and giving me helpful advice.

I would like to thank Dr Paolo Papotti for insightful discussions on data exchange as well as mapping tools Clio, Clip, and +Spicy; Assistant Professor Christopher Scaffidi for answering my questions on Topes; Associate Professor Wang-Chiew Tan and Dr Bogdan Alexe for helping me understand STBenchmark; Dr Wei Wang for helpful discussions on similarity join and its related algorithms; and some members of XQuery WG and XSLT WG including Daniela Florescu, Jerome Simeon, and Michael Kay for giving me advice on the expressiveness and new updates of XSLT and XQuery.

Last but not least, I am forever in debt to my parents. They did everything they could so that I had a good education background.

To my parents, my sister, and my brother. They are my raison d'être

Abstract

Spreadsheets are used by millions of knowledge workers (e.g., accountants, sales persons, project managers, executives, teachers, and programmers) as a routine all-purpose tool for the storage, analysis and manipulation of data. Given the ubiquity and utility of spreadsheets, it has been indispensable to allow data stored in spreadsheets to interact with external applications and Web services. To enable other applications and services to consume or generate spreadsheet data, online spreadsheet applications often provide Web service interfaces (APIs).

In this dissertation, we study the problem of spreadsheet-based data transformation, which transforms spreadsheet data to the structured formats required by external applications and Web services (i.e., *spreadsheet-based data transformation*). We propose a novel framework, namely TranSheet, including methods and tools for supporting both professional programmers and knowledge workers without programming background to transform spreadsheet data to structured formats effectively and easily.

Unlike prior methods, we propose a novel approach in which transformation logic is embedded into a familiar and expressive spreadsheet-like formula mapping language. In terms of expressiveness, popular transformation patterns provided by transformation languages and mapping tools, that are relevant to spreadsheet-based data transformation, are supported in the language via spreadsheet formulas and functions. 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, which allows the mapping to be applied to multiple spreadsheets with similar structure.

Although the reuse of previously specified mappings promises a significant reduction in manual and time-consuming transformation tasks, its potential has not been fully realized in current approaches and systems. To enable users to reuse

previously specified mappings using the above proposed language, we formulate the spreadsheet-based data transformation reuse problem and propose a solution that relies on the notions of spreadsheet templates, mapping generalization, and similarity join. Given a spreadsheet instance that is being mapped to the target schema, we efficiently and effectively recommend a list of previously specified mapping formulas that can be potentially reused for the instance.

In order to make the aforementioned proposed language available to knowledge workers without programming background as well as boost the productivity of professional programmers, we propose a number of novel end-user oriented transformation techniques. We redesign the mapping interface of TranSheet to make it more intuitive and easy-to-use based on nested tables. We develop a collection of form-based transformation operators that help users graphically specify mappings, instead of remembering and writing complex mapping formulas. Users can not only customize an existing operator to suit transformation needs, but also modify a specified transformation operation using a history list. Furthermore, we provide a mechanism for automatically suggesting transformations from source columns to atomic target labels, which is helpful when it is difficult and complicated to specify transformations via formulas or form-based operators.

The approaches proposed in this dissertation have been implemented in prototypes. Moreover, these approaches are validated via experiments in real applications. Real users are also used to evaluate the usability of TranSheet. The experimental results show that : (i) TranSheet is expressive and flexible enough to support numerous practical spreadsheet-based transformation scenarios; (ii) TranSheet is efficient and effective enough to support transformation reuse; (iii) TranSheet significantly reduces transformation specification time and promotes users' satisfaction in comparison with state-of-the-art mapping tools.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Applications and challenges of data transformation | 3 |
| 1.1.1 | Applications of data transformation | 4 |
| 1.1.2 | Challenges of data transformation | 5 |
| 1.2 | Key Research Issues | 7 |
| 1.2.1 | Spreadsheet-based data transformation language | 7 |
| 1.2.2 | Spreadsheet-based data transformation reuse | 9 |
| 1.2.3 | Simplification of spreadsheet-based data transformation | 9 |
| 1.3 | Contributions and Application Scenarios | 10 |
| 1.3.1 | Contributions | 10 |
| | Spreadsheet-like formula mapping language. | 12 |
| | Transformation reuse recommendation. | 13 |
| | End-user centric data transformation techniques. | 14 |
| 1.3.2 | Application scenarios | 15 |
| 1.4 | Thesis Organization | 16 |
| 2 | Background and state-of-the-art on data transformation | 19 |
| 2.1 | Running Example | 19 |
| 2.2 | Basic steps of data transformation | 20 |

| | | |
|----------|---|-----------|
| 2.3 | Schema Matching | 23 |
| 2.4 | Transformation Languages | 25 |
| 2.4.1 | XPath | 26 |
| 2.4.2 | XSLT | 27 |
| 2.4.3 | XQuery | 29 |
| 2.4.4 | Discussion | 30 |
| 2.5 | Mapping systems | 32 |
| 2.5.1 | Clio | 33 |
| 2.5.2 | Clip | 37 |
| 2.6 | Data exchange | 38 |
| 2.7 | String-based data transformation | 40 |
| 2.8 | Transformation by examples | 42 |
| 2.9 | Summary | 44 |
| 3 | Spreadsheet-based data transformation language | 45 |
| 3.1 | Introduction | 45 |
| 3.2 | Data Model | 50 |
| 3.2.1 | Spreadsheet data model | 50 |
| 3.2.2 | Target Data Model | 51 |
| 3.3 | Formula Mapping Language | 52 |
| 3.3.1 | Formal definitions of language's constructs | 53 |
| 3.3.2 | Value mappings | 55 |
| | Copying | 55 |
| | Constant Value Generation | 57 |
| | Derivation | 57 |

| | | |
|-------|---|----|
| | Merging | 58 |
| | Splitting | 58 |
| 3.3.3 | Structural mapping | 58 |
| | Formula inheritance | 59 |
| | Defaults of a structural mapping | 60 |
| | Nesting | 60 |
| | Filtering | 60 |
| | Sorting | 61 |
| | Grouping with aggregation | 61 |
| | Join | 62 |
| | Union/Intersect/Minus | 63 |
| | Branching | 64 |
| | Composition and refinement of mappings | 65 |
| 3.3.4 | User-defined functions | 65 |
| 3.4 | Target Schema Restructuring | 68 |
| 3.4.1 | Isomorphic view rearrangement | 68 |
| | Label reordering | 68 |
| | Ignoring and adding labels | 69 |
| 3.4.2 | Anisomorphic view rearrangements | 70 |
| 3.4.3 | Specializing the multiplicity of repeating labels | 71 |
| 3.4.4 | Specializing choice constructs | 72 |
| 3.5 | Generalizing mapping formulas | 73 |
| 3.5.1 | Generalization using native spreadsheet formula functions . . . | 73 |
| 3.5.2 | New notations for generalizing mappings | 74 |
| | Specifying relative location of spreadsheet data | 75 |

| | |
|--|------------|
| Dynamic Range Length | 76 |
| Mapping of non-adjacent collections of cells | 77 |
| 3.6 Mapping formula interpretation | 78 |
| 3.6.1 TGD Generation | 78 |
| 3.6.2 Query Generation | 84 |
| Basic steps and properties of query generation | 84 |
| Novelty in query generation | 85 |
| 3.7 Implementation | 87 |
| 3.8 Experiments | 90 |
| 3.8.1 Expressiveness | 90 |
| 3.8.2 Mapping generalization | 94 |
| 3.9 Related Work | 96 |
| 3.9.1 Transformation Languages and Mapping Tools | 96 |
| 3.9.2 Exportation of spreadsheet data | 97 |
| 3.9.3 Schema Mapping and Data Exchange | 97 |
| 3.9.4 Spreadsheet-based data access and manipulation | 98 |
| 3.9.5 Domain specific and data description languages | 98 |
| 3.10 Summary | 99 |
| 4 Spreadsheet-based Data Transformation Reuse | 101 |
| 4.1 Introduction | 101 |
| 4.2 Problem Definition | 105 |
| 4.3 Spreadsheet Template | 106 |
| 4.3.1 Template Description Language | 106 |
| 4.3.2 Inferring Templates | 108 |

| | | |
|----------|--|------------|
| 4.4 | Reuse Recommendation Algorithm | 110 |
| 4.5 | Implementation | 116 |
| 4.5.1 | Architecture | 116 |
| 4.5.2 | Mapping Repository Organization | 116 |
| 4.6 | Evaluation | 118 |
| 4.6.1 | Performance | 118 |
| 4.6.2 | Effectiveness | 121 |
| 4.7 | Related Work | 123 |
| 4.8 | Summary | 125 |
| 5 | End-user oriented spreadsheet-based data transformation | 126 |
| 5.1 | Introduction | 127 |
| 5.1.1 | Running Example | 127 |
| 5.1.2 | Contributions | 129 |
| 5.2 | User Interface | 130 |
| 5.2.1 | Mapping Sheet | 130 |
| 5.2.2 | Matching Sheet | 132 |
| 5.3 | Form-based Transformation Operators | 133 |
| 5.3.1 | Transformation Operator Definition | 134 |
| 5.3.2 | Transformation Operator Customization | 135 |
| | Customization Operator Definition | 136 |
| | Customization Operator Generation | 137 |
| 5.3.3 | Design of Transformation Operators | 139 |
| | Transformation Operators for Value Mappings | 139 |
| | Transformation Operators for Structural Mappings | 141 |

| | | |
|----------|---|------------|
| 5.3.4 | Transformation Operation Modification | 145 |
| 5.3.5 | Expressiveness | 146 |
| 5.4 | Automated Transformation Suggestions | 147 |
| 5.5 | Implementation | 149 |
| 5.6 | User study | 150 |
| 5.6.1 | Experimental Setup and Methodology | 151 |
| 5.6.2 | Observations | 152 |
| 5.6.3 | Post-study Questionnaire | 155 |
| 5.7 | Related Work | 155 |
| 5.8 | Summary | 158 |
| 6 | Conclusion and Future Work | 159 |
| 6.1 | Concluding Remarks | 159 |
| 6.1.1 | An expressive and familiar spreadsheet-like formula mapping language | 160 |
| 6.1.2 | Previously specified mapping formulas reuse recommendation . | 162 |
| 6.1.3 | End-user oriented spreadsheet-based transformation techniques | 163 |
| 6.2 | Future Directions | 164 |
| | Publications | 167 |
| A | Appendix | 168 |
| A.1 | Template inference based on cp-similarity | 168 |
| A.2 | TranSheet’s screenshots | 170 |
| | Bibliography | 175 |

List of Figures

| | | |
|-----|--|-----|
| 1.1 | Global architecture of TranSheet | 11 |
| 2.1 | Running Example | 20 |
| 2.2 | Basic steps of data transformation | 21 |
| 2.3 | Mapping interface of commercial tool Altova MapForce for implementing the running example | 34 |
| 2.4 | Simultaneous editing interface of PotLuck | 42 |
| 3.1 | (a) The Swine Flu data set; (b) <i>Pie Chart</i> schema; (c) <i>Scatter Plot</i> schema; (d) <i>Bar Chart</i> schema | 46 |
| 3.2 | A spreadsheet and its mapping specification | 56 |
| 3.3 | A tabular representation of orders and its corresponding mapping | 59 |
| 3.4 | A generalized mapping specification for exporting datasets of varying sizes | 73 |
| 3.5 | Architecture of TranSheet for the spreadsheet-based data transformation language | 89 |
| 3.6 | TranSheet user interface | 95 |
| 4.1 | Motivating example: (a) Employees organized in a table presentation; (b) Employees organized in a vertical repeater presentation without address information; (c) Employees organized in a vertical repeater presentation with address information; (d) The target schema. | 103 |

| | | |
|------|---|-----|
| 4.2 | Inferred template of the instance in Figure 4.1(b) in a new worksheet | 110 |
| 4.3 | TranSheet architecture for spreadsheet-based data transformation reuse | 117 |
| 4.4 | Mapping Repository Organization | 118 |
| 4.5 | Performance graph of Jaccard similarity. | 119 |
| 4.6 | Performance graph of Cosine similarity. | 120 |
| 4.7 | Reuse effectiveness of Jaccard similarity. | 122 |
| 4.8 | Reuse effectiveness of Cosine similarity. | 122 |
| 5.1 | Running example: (a) Source spreadsheet; (b) Target schema | 128 |
| 5.2 | Mapping interface of a relationship-based mapping tool | 128 |
| 5.3 | Side-by-side comparison: (a) Source spreadsheet data; (b) Target data represented using a nested table | 132 |
| 5.4 | Merging Operator | 134 |
| 5.5 | (a) Editing Mode of Filtering Operator; (b) Filtering Operator | 138 |
| 5.6 | Splitting Operator | 141 |
| 5.7 | Join Operator | 143 |
| 5.8 | Transformation Operators: (a) Grouping Operator; (b) Aggregate Operator | 143 |
| 5.9 | Sorting Operator | 144 |
| 5.10 | Architecture of TranSheet for spreadsheet-based data transformation simplification | 150 |
| 5.11 | User study completion time (mean) | 152 |
| 5.12 | User study completion time (median) | 153 |
| A.1 | (a) Sales data; (b) Mapped target schema | 169 |
| A.2 | Automatically Inferred Template | 169 |
| A.3 | Interface for mapping swine flu data to a pie chart | 170 |

| | | |
|-----|---|-----|
| A.4 | Schema repository of TranSheet | 171 |
| A.5 | Interface for mapping swine flu data to a pie chart with the selected target schema | 172 |
| A.6 | Interface for mapping swine flu data to a pie chart with instant feed- back for mappings | 172 |
| A.7 | Transformation preview of TranSheet | 173 |
| A.8 | Interface for mapping swine flu data to a scatter plot | 173 |

List of Tables

| | | |
|-----|---|-----|
| 2.1 | List of the semantic correspondences produced by COMA++ for the running example | 24 |
| 3.1 | Transformation patterns of TranSheet | 53 |
| 3.2 | Mapping Scenarios | 91 |
| 3.3 | Source manipulation operations of EXM and ManyEyes in implementing mapping scenarios | 93 |
| 4.1 | Tokens' document frequencies and token's orders of P_{I_0} , P_{I_1} , and P_J (Part 1) | 113 |
| 4.2 | Tokens' document frequencies and token's orders of P_{I_0} , P_{I_1} , and P_J (Part 2) | 113 |
| 5.1 | Matching sheet for the running example | 133 |
| 5.2 | Two tables need to be joined | 142 |
| 5.3 | Users' rating results | 155 |

Chapter 1

Introduction

Information integration is the problem of combining information from different sources into a unified format, which is considered as a major challenge faced by modern organizations [49]. Two specific facets of information integration are data exchange and data integration. Transformation of data from one schema to another is frequently necessary when there is a need of integrating or exchanging data between independently developed applications [84, 94, 71, 105].

Data transformation converts data structured under one schema (the source schema) into data structured under another schema (the target schema) [74]. It has been recently renewed with the increasing explosion of relational data, XML files, and office documents (e.g., emails, spreadsheets, and text documents) on the Web, within enterprises, and in large-scale scientific projects [36, 109, 86].

In the last several years, there has been a rich body of research in terms of theoretical foundations and practical tools on data transformation [10, 111, 127]. Matching techniques [119, 44, 64, 108, 50] (semi)-automatically help users find semantic correspondences between the source and target schemas. Users deal with transformations at a higher level of abstraction using the graphical user interfaces (GUIs) of mapping tools [127], instead of writing difficult and error-prone programs using powerful transformation languages (e.g., XSLT [17], SQL/XML [66], XQuery [16]). Besides research prototypes Clio [85, 116, 79], Clip [117], and +Spicy [112] with solid theoretical foundations [100, 71], there has been a proliferation of industry

mapping tools (e.g., IBM Relational Data Architect [7], Altova MapForce [1], Stylus Studio [9], MS BizTalk Mapper [3]).

However, data transformation is still a labour-intensive, time-consuming, and cumbersome task [85]. Mainstream 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 [72, 143]. 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 [136, 136, 5] 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. Moreover, these solutions are mainly developed with an enterprise setting, which requires the expertise of professional programmers. In addition to that, these solutions mainly focus on the development of an ad-hoc program that can handle only exactly one data source, without an explicit intent for future reuse [128]. In other words, past transformation efforts are not effectively leveraged to save time and avoid effort duplication.

Spreadsheets are ubiquitous tools used for the storage, analysis and manipulation of data [133]. There are several reasons for their popularity. Spreadsheet-based data management offers important flexibility in data formatting over a tabular grid [34]. 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 [96]. 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, sales persons, teachers, project managers, executives, and programmers), who play a key role in critical enterprise activities [93].

Given the ubiquity and utility of spreadsheets, it is increasingly necessary to allow data stored in spreadsheets to interact with external applications and ser-

vices [114, 125]. One of the key problems is transforming spreadsheet data to the structured formats required by these services and applications [60]. There has been a proliferation of online spreadsheet-like applications including Google Spreadsheets [6], Excel Web App [4], and Zoho Spreadsheet [22]. To enable other applications to consume or generate spreadsheet data, some of these applications provide Web service interfaces (APIs). The authors in [93] report that spreadsheets often serve as hubs for organizing and manipulating information, which is later transferred to other services for archiving or processing.

This dissertation studies the problem of spreadsheet-based data transformation, which transforms spreadsheet data to the structured formats required by external applications and Web services (i.e., *spreadsheet-based data transformation*). 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. We particularly aim at enabling both professional programmers and knowledge workers, who do not possess programming background, to effectively, easily, and conveniently transform spreadsheet data to structured formats.

This chapter is organized as follows. Section 1.1 provides an overview of data transformation including its applications and some main challenges that make data transformation cumbersome, time-consuming, and labor-intensive. Next, Section 1.2 outlines the research issues tackled in this dissertation. Afterwards, Section 1.3 summarizes our main contributions and presents some applications of our work. Finally, Section 1.4 gives the roadmap for the whole dissertation.

1.1 Applications and challenges of data transformation

In this section, to motivate the importance of data transformation, we summarize its usefulness in several application domains in Section 1.1.1. Section 1.1.2 presents some main challenges of data transformation.

1.1.1 Applications of data transformation

Data Exchange

Data exchange (i.e., data translation) is basically process of transforming an instance of a given source schema to an instance of a given target schema according to the relationship between the source and target schemas [71, 100, 88, 73]. In particular, data exchange entails the materialization of data, after the data have been extracted from the source and restructured into the unified format.

Data Integration

The main objective of data integration is to provide users with a uniform interface (i.e. mediated schema or global schema) to query multiple autonomous and highly diverse data sources, therefore shielding users from locating and interacting with each data source, and then manually combining results [105, 86]. Data integration systems formed a new industry, namely Enterprise-Information Integration (EII) [87]. Data sources must be transformed to match with the global schema (target schema) as part of semantic query processing. It is worth noting that, unlike data exchange, an instance of the target schema is just a virtual view, not being materialized.

SOA

Service-Oriented Architecture (SOA) enables organizations to quickly respond to ever changing business requirements by reusing and leveraging existing applications and resources in which their functionality is exposed via XML-based Web services. SOA largely focuses on developing composite applications [39, 55]. To develop a composite application, it is necessary to transform the messages of constituent applications into the internal format required by the composite one. Together with workflow technology and traditional Enterprise Application Integration (EAI) systems, SOA is one of the main approaches of application integration.

Data Warehousing

A data warehouse is a decision support database that consolidates data from multiple data sources [45]. Data coming from external data sources must be transformed to the internal representation (schema) of data warehouse. External data may be overlapped and needs to be cleaned to remove duplicates. Extract-Transform-Load (ETL) tools are used to address these problems, including a collection of cleaning operations (e.g., detection of approximate duplicates) and transformation operations [99].

Data Cleaning

Data cleaning (i.e., data cleansing or scrubbing) is the process of detecting and removing errors (e.g., misspellings and invalid data) and inconsistencies (e.g., duplicate information) from data to improve the quality of data [120]. Data cleaning is typically performed together with data transformation [121]. Data is first analyzed to find errors and then suitable transformations are applied to fix them.

1.1.2 Challenges of data transformation

Despite its pervasiveness and importance, data transformation is still a cumbersome and time-consuming task. Given the source (S) and target (T) schemas, and an instance I of S, data transformation produces a target instance J conforming to T according to a specified mapping \sum_{ST} between S and T. This problem is challenging because of the following reasons:

Schema matching

Before a mapping specification begins, it is first needed to establish semantic correspondences between the elements of S and T, which is also termed *schema matching*, *ontology matching*, and *ontology alignment* in the literature [108, 119, 64, 44]. For example, elements **FirstName** and **LastName** of S correspond to element **Name** of T.

The semantics of the source and target elements can be only inferred from a few

sources, including creators of data, related documentation, and associated schemas and data [65]. Unfortunately, data creators may have retired or forgotten about the data; related documentation may be sketchy, outdated, or unavailable.

Thus, the major information source relies on the clues of schemas and data, such as element names, types, data values, schema structures, and integrity constraints. However, these clues may be unreliable and insufficient. For example, two elements with the same name can refer to different real-world entities (e.g., **area** can refer to either the location or square-feet area). Conversely, two elements with different names can refer to the same entity (e.g., **area** and **location** both refer to the location of a house).

Furthermore, matching may be subjective and depends on application domains. As a result, users often need to involve themselves in matching processes. Sometimes, the involvement of a single user is considered subjective and a large group of people must participate in decision making activities [65].

Mapping specification

Writing transformations using powerful transformation languages (such as SQL/XML [66], XSLT [12], and XQuery [16]) by hand is very difficult, tedious, and error-prone. As a result, mapping technology has been developed to help automate this process [127]. A visual mapping tool is used to specify at higher and language-neutral level how to transform the source instance *I* to the target instance *J*. The mapping specification is then translated into executable code (e.g., XSLT, XQuery, Java, C#), which can be later deployed to a runtime engine (e.g., Saxon [26], AltovaXML [40], and Xalan [78]) for transformation execution.

In regard to mapping interface, most mapping tools follow *relationship-based metaphor*, in which the source schema (*S*) is located on the left side of the screen and the target schema (*T*) is located on the right side of the screen [127]. Lines are connected between the elements of the two schemas and these lines may be annotated with one or more functions to specify a complex relationship. This flow-chart-like interface is typically cluttered and unintuitive for knowledge workers as pointed

out in the literature [126, 19, 113, 140]. This may be aggravated if the number of elements of the source and target schemas are large and there are multiple lines with annotated functions connecting between the elements of the two schemas.

In addition to that, the built-in functions provided by these mapping tools are unfamiliar to knowledge workers without programming background since these functions are similar to the ones of low-level programming languages. For example, Altova Mapforce's functions [1] are based on the ones of XPath, XSLT, XQuery, Java, C#, C++; the functions of MS BizTalk Mapper [3] are powered by the programming languages (e.g., C# and VB.NET) of .NET framework [61]; the functions of Stylus Studio [9] are identical the ones of XSLT, XQuery, and Java.

Moreover, these mapping tools separate between two modes, namely *development mode* and *execution mode*. Like professional programming environments, users must compile a mapping specification to see transformation result. This raises another hassle because instant feedback is not available for each step of the mapping specification.

Last but not least, the transformed target instance and the source instance are located in two separate windows, which make comparison between these two instances cumbersome. As a result of that, transformation validation and refinement often take time.

1.2 Key Research Issues

In Section 1.1.2, we outlined some key challenges of data transformation. In this section, we present several specific research issues in the context of spreadsheet-based data transformation that will be tackled in this dissertation.

1.2.1 Spreadsheet-based data transformation language

Given the fact that a significant amount of the world's data is maintained in spreadsheets, it has been indispensable for using data stored in spreadsheets to interact with external applications and Web services [114, 125]. We consider the problem

of developing a language for transforming spreadsheet data to the structured formats required by these applications and services. This problem is different from traditional data transformation [71] because of: (i) the characteristics of the source data model (spreadsheets); (ii) the users that we aim at, who are familiar with the spreadsheet programming paradigm.

This problem is challenging because of the nature of spreadsheet documents. First, the data spreadsheets contain is not organized following a predefined schema due to the flexibility of spreadsheets in terms of data formatting. Second, there may be mismatches between the organization of spreadsheet data and the structure expected by an external application. For example, while spreadsheet data is organized in a table, the schema used by a bar chart (used by ManyEyes [136]) consists of a list of labels, each comprising a list of bars.

The language should meet the following requirements. First, the language should leverage existing users' programming experience (e.g., spreadsheet-like formulas). This ensures that the investment made by users in learning spreadsheet programming is paid off. Second, the language should preserve the important characteristics of the spreadsheet programming paradigm (e.g., instant feedback at each step) in order to make mapping specification productive. Third, the language should provide utilities to help users work around structural mismatches between spreadsheet data organization and the target schema. Fourth, the language should be expressive enough to support numerous spreadsheet-based transformation scenarios compared with other popular transformation languages (e.g., XSLT and XQuery) and mapping tools [37]. Finally, since a spreadsheet may be mapped to multiple target schemas (e.g., visualize a data set using multiple visualization types), the language should preserve the presentation of the spreadsheet when specifying a mapping. As a result, users do not have to tediously and laboriously modify a spreadsheet to specify mappings as well as maintain multiple versions of the spreadsheet.

1.2.2 Spreadsheet-based data transformation reuse

Given that data transformation is a labour-intensive and error-prone process [85], it is useful to reuse previously specified mappings as much as possible to save time and reduce effort duplication. We consider problem of reuse in transforming spreadsheet data to the structured formats required by external applications and Web services.

The problem is challenging because: (i) Spreadsheet systems do not impose many constraints on spatial layout of data, and users can organize same data according to their own preferences and styles, not in a pre-defined way. Therefore, given two spreadsheet instances, it is programmatically difficult to uncover if they are similar in terms of structure. Mapping a spreadsheet instance to a target schema typically depends on the spatial layout of the instance; (ii) Given a spreadsheet instance and a target schema, there may have multiple ways of mapping the instance to the schema. For instance, an optional attribute of the target schema can be mapped or not mapped to spreadsheet data; (iii) A mapping of a spreadsheet instance to the target schema is only applied exactly to this instance, not to other instances with similar structure that also need to be mapped to the target schema; (iv) Since a mapping repository may contain a large number of mappings (up to a few hundred thousand mappings), the reuse recommendation mechanism should suggest previously specified mappings in an effective and efficient way.

1.2.3 Simplification of spreadsheet-based data transformation

On one hand, mainstream data transformation solutions mainly target professional programmers. Consequently, it is still unintuitive and difficult for knowledge workers without programming background to specify transformations. On the other hand, data transformation has been increasingly necessary for knowledge workers to analyze, manipulate and visualize data (e.g., social data analysis [137], Web mashup [139, 142], and SOA [39]).

Although the language we mentioned in Section 1.2.1 leverages users' spreadsheet

programming experience, we mainly focus on the foundations and semantics of the language, rather than its usability aspects.

We consider the problem of simplification in transformation spreadsheet data to structured formats, which makes spreadsheet-based data transformation available to non-technical users. Also, professional programmers can boost their productivity by using more usable techniques. The main challenges are: (i) to provide an intuitive and familiar interface for transformation specification, instead of relying on the cluttered and unintuitive interface of relationship-based mapping tools [127]; (ii) to provide utilities that enables the user to specify transformations via the reuse and customization of pre-defined transformation operators, rather than remembering complex syntax and writing transformation programs from scratch; (iii) to automatically suggest transformations from source columns to target labels. This is helpful when specifying transformations using the language mentioned in Section 1.2.1 or the above pre-defined transformation operators is complicated and difficult.

1.3 Contributions and Application Scenarios

We summarize the main contributions of the dissertation in Section 1.3.1. We then describe some real application scenarios of our work in Section 1.3.2.

1.3.1 Contributions

To address the research issues outlined in Section 1.2, we develop a novel framework, namely TranSheet, with the global architecture depicted in Figure 1.1. In a nutshell, this framework enables users to: (i) transform spreadsheet data to structured formats using a familiar and expressive spreadsheet-like formula mapping language; (ii) reuse previously specified mappings effectively and efficiently to save time and reduce effort duplication; (iii) specify transformations in an intuitive and easy-to-use way without requiring deep programming background. We believe that our framework will benefit both professional programmers and knowledge workers without programming background.

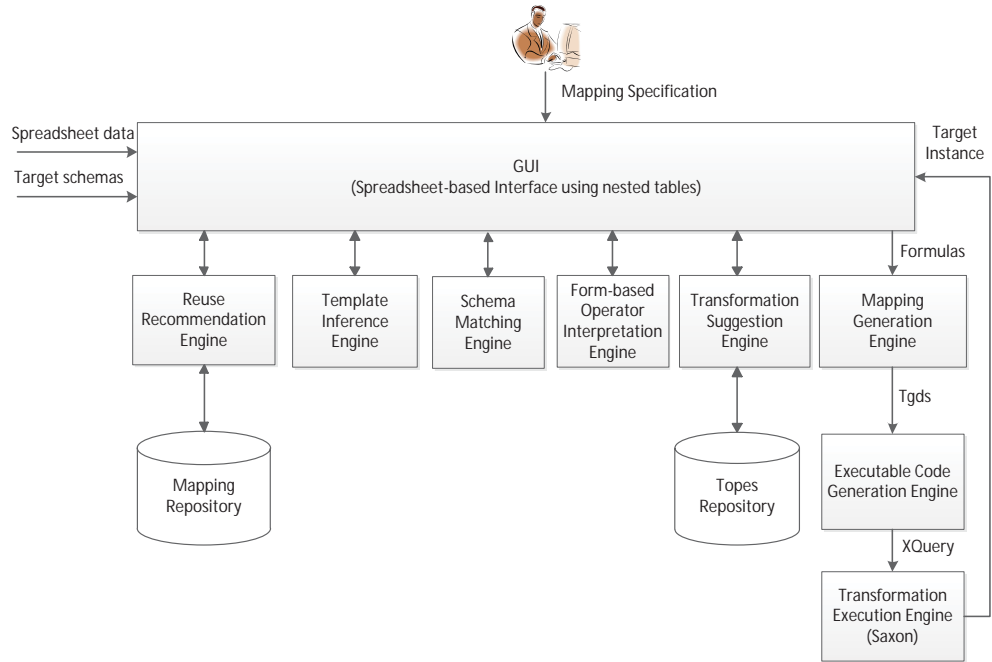


Figure 1.1: Global architecture of TranSheet

We choose MS Excel as the main spreadsheet environment for development due to its ubiquity. However, the concepts presented in this dissertation are generic and, therefore, are applicable to other spreadsheet environments as well (e.g., OpenOffice [25], Gnumeric [28], Google Spreadsheets [6], and Apple Numbers [20]).

To achieve the aforementioned goals, we first build the foundations of TranSheet by developing a spreadsheet-like formula mapping language that takes into account the expressiveness in comparison with popular transformation languages [17, 16] and mapping tools [38]. We then provide users with facilities to effectively and efficiently reuse previously specified mapping formulas by extending this language. Finally, we propose some novel end-user oriented transformation techniques so that knowledge workers without programming background can make the most of the language by specifying mappings easily and professional programmers can boost their productivity with the language. In this dissertation, we make a number of unique and significant contributions which are summarized in the following.

Spreadsheet-like formula mapping language.

We propose a novel approach, which enables users to perform mappings via a familiar and expressive spreadsheet-like formula language. Our work differs from existing approaches on spreadsheet-based data transformation [60, 123, 136] in the sense that: (i) we leverage existing users' experience in spreadsheet programming and preserve important characteristics of spreadsheet programming; (ii) we exploit frequently used formatting features of spreadsheets to generalize a mapping from instance level to template level; (iii) we avoid cluttering spreadsheet documents with transformations by embedding transformation logic into the language. More specifically, the main contributions are as follows:

- A spreadsheet-like formula language is designed for specifying mappings between spreadsheet content and the target schema. In terms of *expressiveness*, we demonstrate that popular transformation patterns that are relevant to spreadsheet-based transformation are supported in the language using spreadsheet formulas and functions. These patterns are the result of a careful analysis of commonly needed mapping scenarios supported by transformation languages (e.g., XSLT/XQuery), mapping tools [37], and spreadsheet corpora [77]. This enables the language to avoid cluttering the source spreadsheet with transformations and it turns out to be helpful when multiple target schemas are mapped (e.g., visualize a data set using multiple visualization types).
- *Target schema restructuring* is proposed to allow users to resolve the mismatches between the source spreadsheet and the target schema. It is a common occurrence that the target schema, which is defined externally, does 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 how mapping formulas should be interpreted.
- The language supports the generalization of a mapping from instance-level to template-level element that allows applying the mapping for multiple instances

with similar structure. Formatting features frequently used in spreadsheets templates (e.g., relationships between cells, cell styles, and border styles) are exploited to generalize mappings. As a result of that, TranSheet is able to transform a large number of naturally occurring spreadsheets belonging to this class, which cannot be handled by the alternative approaches.

- We use *tuple generating dependencies* (*tgds*) [72, 143], a widely used schema mapping formalism, to describe the semantics of TranSheet. In comparison with the state-of-the-art, we introduce a collection of new functions to *tgds* expressions. We then extend a previous query generation algorithm [79, 118] to generate executable queries (i.e., XQuery) for these functions. Consequently, each target document generated by TranSheet corresponds to a canonical universal solution of data exchange [71].
- We implemented a prototype of the language 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 data transformation scenarios.

Transformation reuse recommendation.

We extend the above language (Section 1.3.1) to address the problem of spreadsheet-based data transformation reuse. We formulate the problem and propose a solution that relies on the notions of spreadsheet templates, mapping generalization, and similarity join. Given a spreadsheet instance that is being mapped to the target schema, we recommend a list of previously specified mapping formulas that can be potentially reused for the instance. To the best of our knowledge, the problem of spreadsheet-based transformation reuse has not been addressed before in the setting we consider here. More specifically, we make the following contributions:

- We formulate the problem of spreadsheet-based transformation reuse as a variant of *similarity join* [43, 57, 141], which is a well-known similarity search problem that finds all pairs of objects whose similarity is above a given threshold.

- We define spreadsheet templates that are used to characterize spreadsheet structures. We propose techniques to infer a template from an existing spreadsheet based on common spreadsheet presentation patterns. We then generate the string-based representation of an inferred template.
- We propose an algorithm to recommend previously specified mappings for a new spreadsheet instance that needs to be mapped to the target schema. This relies on computing similarity between string-based representations of templates.
- We design a repository to organize mapping information. We implemented a prototype of the proposed solution and evaluated its performance. The experimental results show that our solution is efficient enough to support a few hundred thousand mappings stored in the mapping repository. TranSheet is also effective enough to support transformation reuse.

End-user centric data transformation techniques.

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 formula mapping language described in Section 1.3.1 may be complex for them. Furthermore, even expert users, who are already familiar with the syntax of the language, sometimes wish to boost their productivity by not having to remember and write complex syntax.

To tackle the challenges in simplifying spreadsheet-based data transformation presented in Section 1.2.3, we make the following contributions:

- The mapping interface is designed based on nested tables, which is more intuitive and easy-to-use for non-technical users. With this new interface, users can preview the whole transformation result right in the sheet containing source data. This is convenient for side-by-side comparison in order to validate and refine transformations. Besides, a matching module is integrated to help users

semi-automatically find semantic correspondences between the source spreadsheet and the target schema.

- A set of form-based transformation operators are proposed allowing users to specify mappings graphically. The benefits of these operators are two-fold: (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 in spreadsheet programming. We define these operators in a generic way in order to cover numerous transformation patterns. We provide a form customization mechanism allowing users to customize an existing operator to suit transformation needs. We also offer a history list allowing users to modify specified transformation operations.
- TranSheet automatically suggests transformations from source columns to atomic target labels. This relies on employing *Topes* [132], where each Tope is a category of data with different formats and functions for transforming between these formats. Automated transformations are helpful when specifying transformations using formulas or form-based transformation operators is complicated and difficult.
- We implemented a prototype and conducted an extensive user study across a set of real spreadsheet-based transformation tasks to evaluate the usability of our approach. The experimental results show that TranSheet significantly reduces specification time and promotes users' satisfaction in comparison with state-of-the-art mapping tools.

1.3.2 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. From Office 2007, Microsoft uses Office Open XML (OpenXML) as the default format for data storage instead of the binary file formats [125]. Recently, OpenXML is approved by International Organization for Standardization (ISO) as an international standard. Due to the ubiquity of Excel,

this will further facilitate the exchange of spreadsheet data with other applications and services. In the following, we describe some real application scenarios. They are only part of an almost unlimited list of possibilities.

Business users can use TranSheet to interact with the enterprise business systems 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 [136] and Google Fusion Tables [5]. 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.

1.4 Thesis Organization

The remainder of the dissertation is organized as follows. In Chapter 2, we provide a background as well as the state-of-the-art of data transformation. The main objective is to review the literature on data transformation and uncover its shortcomings. We first describe the basic steps of a transformation task. We then briefly present key schema matching techniques, which is not the main focus of this dissertation. Next, popular transformation languages and visual mapping tools are discussed in details with examples. Data exchange theory, which is a generalization

of the mapping generation algorithms of *tgds*-based mapping tools, is presented after that. We also look at string-based data transformation and transformation by examples techniques. Finally, we summarize and point out some main drawbacks of the state-of-the-art.

In Chapter 3, we lay the foundations of TranSheet by developing a spreadsheet-like formula mapping language that enables users to express mappings between spreadsheet data and the elements of the target schema. We first characterize the problem by defining the data models of spreadsheets and the target schema. We then explain the main constructs of the language in details. Target schema restructuring is presented after that. Next, we describe the mapping generalization constructs, allowing a mapping to be applied to a set of spreadsheet instances with similar structure. Afterwards, we formally present the semantics of the language using *tgds*. We then describe how to generate executable code (i.e., XQuery) from produced *tgds*. Finally, we present the prototype implementation and evaluate the expressiveness and mapping generalization of TranSheet in two real applications.

In Chapter 4, we study the problem of reuse in transforming spreadsheet data to structured formats. First, we formulate this problem as a variant of a similarity join. Next, we define spreadsheet templates, which are used to characterize spreadsheet structures. We then propose techniques to infer templates from existing spreadsheets. We then generate the string-based representation of an inferred template. Afterwards, we propose an algorithm to recommend previously specified mappings for a new spreadsheet instance that needs to be mapped to the target schema. We then design a repository to organize mapping information. Finally, we implement a prototype of the proposed solution and evaluate its performance and effectiveness.

In Chapter 5, we propose some novel techniques to simplify spreadsheet-based data transformation. We first redesign the mapping interface based on nest tables to make mapping specification more intuitive, and then discuss how a matching module is integrated. Next, we present form-based transformation operators (including operator definition, operator customization, operator interface design and semantics, transformation operation modification, and the expressiveness of operators) that help users specify mappings graphically, instead of writing complex mapping

formulas from scratch. Afterwards, we describe automated transformation suggestions supported by TranSheet. Finally, we discuss the prototype implementation and conduct an extensive user study to evaluate the usability of our techniques.

Finally, Chapter 6 gives concluding remarks and discusses possible directions for future work.

Chapter 2

Background and state-of-the-art on data transformation

In this chapter, we focus on providing central concepts in data transformation and then reviewing the state-of-the-art. The chapter is structured as follows. We first describe the running example in Section 2.1 that is used throughout the chapter for illustration purpose. We present the basic steps of a data transformation task in Section 2.2. Afterwards, we explain the key techniques of schema matching in Section 2.3. We then describe two popular transformation languages, namely XSLT and XQuery, in Section 2.4. Next, some major mapping systems are outlined in Section 2.5. Data exchange theory is briefly discussed in Section 2.6. We also look at some work on string-based data transformation and transformation by examples techniques in Sections 2.7 and 2.8, respectively. Finally, we summarize and describe some limitations of the state-of-the-art in Section 2.9.

2.1 Running Example

We now describe the running example depicted in Figure 2.1 for this chapter. There is a new incoming order (Listing 2.1) conforming to the source schema shown on the left side in Figure 2.1 that must be transformed to the internal format conforming to the target schema shown on the right side in Figure 2.1. More specifically, the

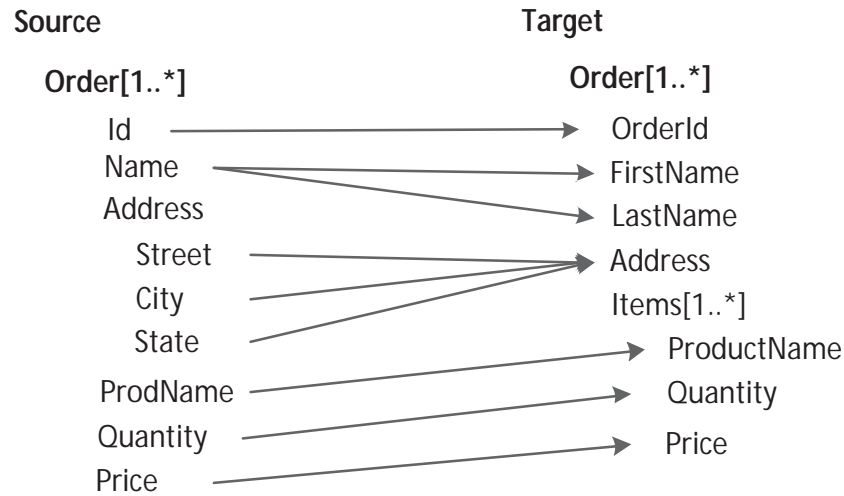


Figure 2.1: Running Example

following mappings are performed:

- Values of **Id** are copied to values of **OrderId**. Similarly, values of **ProdName** and **Quantity** of the source are copied to values of **ProductName** and **Quantity** of the target, respectively.
- Values of the source element **Price** are converted to formats expected by values of the target element **Price** (e.g., convert Australia dollar to US dollar)
- Split values of element **Name** into values of **FirstName** and **LastName** according to delimiter whitespace “ ” in the middle of values of element **Name**.
- Merge values of elements **Street**, **City**, and **State** into values of **Address** with delimiters whitespace “ ” between elements **Street** and **City**, **City** and **State**.

2.2 Basic steps of data transformation

In order to review existing approaches on data transformation, we describe the basic steps of a data transformation task shown in Figure 2.2. First, semantic *correspon-*

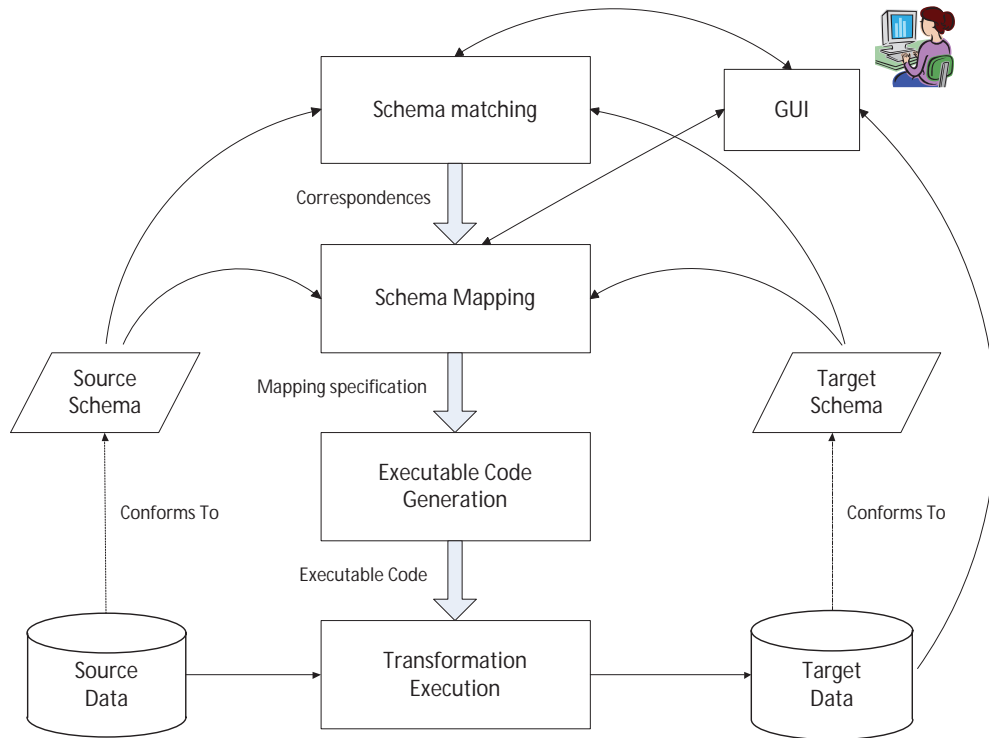


Figure 2.2: Basic steps of data transformation

dences must be identified between elements of the source and target schemas (e.g., element **Id** corresponds to element **OrderId**; element **Name** corresponds to elements **FirstName** and **LastName**; elements **Street**, **City**, and **State** correspond to element **Address**). This is done via a *schema matching* engine [119, 108, 44, 64].

Second, it is needed to precisely specify how to relate the source instances to the target instances according to the above identified correspondences, which is called *schema mapping*. Such correspondences strongly affect quality of a mapping system: faulty correspondences lead to invalid mappings, and therefore producing incorrect target instances [51]. Schema mapping can be seen as interpretations with runtime consequences of the correspondences that result after schema matching [85]. For example in Figure 2.1, while schema matching only suggests that **FirstName** and **LastName** in one schema are related to **Name** in the other schema, schema mapping says that concatenating the former yields the later (e.g., $\text{Name} = \text{FirstName} + " " + \text{LastName}$) [50].

Then, executable code (e.g., Java, C#, SQL, XSLT, and XQuery) is generated from a mapping specification to implement data transformation via a code generation engine. Note that some transformation systems may generate intermediate code for mapping optimization, reuse, and manipulation (e.g., *mapping composition* and *mapping inversion* [143]).

Next, the source instance is transformed to the target instance by executing the generated code using an execution engine (e.g., AltovaXML [40] and Saxon [26]).

Finally, the target instance is shown up to users for transformation verification and refinement. Note that schemas and mappings can be stored in a repository for future reuse.

Listing 2.1: An instance of the source schema of the running example stored in file `Orders.xml`

```
<?xml version="1.0"?>
<Source>
  <Order>
    <Id>42</Id>
    <Name>Ford Prefect</Name>
    <Address>
      <Street>Addison</Street>
      <City>Sydney</City>
      <State>NSW</State>
    </Address>
    <ProdName>Beer</ProdName>
    <Quantity>3</Quantity>
    <Price>1.5</Price>
  </Order>
  <Order>
    <Id>525</Id>
    <Name>Arthur Dent</name>
    <Address>
      <Street>Evans</Street>
      <City>Sydney</City>
      <State>NSW</State>
    </Address>
    <ProdName>Beer</ProdName>
    <Quantity>4</Quantity>
    <Price>2</Price>
  </Order>
</Source>
```


2.3 Schema Matching

The main goal of schema matching is to identify semantic correspondences between the source and target schemas (i.e., identify similar elements between two schemas). For example, in Figure 2.1, `Id` is semantically related to `OrderId`, while it has no semantic relationship with `FirstName`. Schema matching is a difficult task to automate since only schema designers fully understand semantics of schema elements and design documentation is often poor and cannot capture completely semantics [65]. One solution is a human expert manually provides correspondences after thoroughly investigating the source and target schemas. However, when the structures of the source and target schemas are complex and the number of elements is large, such manual process is very time-consuming and labor-intensive.

To increase productivity, a schema matching module can be used to semi-automatically return a set of correspondences. Matching is often performed at either schema level or instance (data contents-based) level [119]. Various types of information at schema level can be exploited to find correspondences. Some of them are as follows:

- *Element name-based matching* matches elements with equal names (e.g., `Price` vs `Price` and `Quantity` vs `Quantity`) or similar names (e.g., `ProdName` vs `ProductName` and `Id` vs `OrderId`).
- *Structural level matching* refers to matching combinations of elements appearing together in a structure. For example, `Source.Order` partially structurally matches `Target.Order`.
- *Constraint-based matching* uses constraints on data types, value ranges, uniqueness, and so on to identify the similarity of elements. For example, `ProdName` and `ProductName` have the same type string; `Id` and `OrderId` are both unique keys in the source and target schemas.

Each *matcher* (i.e., a matching algorithm) uses the above information to solve a given match task. While a *hybrid matcher* directly combines multiple information to determine matching candidates (e.g., name-based matching combines with

data type-based matching, namely name-type matcher, to identify the correspondence `ProdName` vs `ProductName`), a *composite matcher* combines results of several independently executed matchers including hybrid matchers (e.g., first use type-name matcher and then use name-path matcher [64]). The result of executing each matcher is usually a similarity matrix, in which candidates are entries of the matrix with highest similarities. This produces different matching cardinalities 1-1 (e.g., `Id` vs `OrderId` in Figure 2.1), 1-n (e.g., `Name` vs `FirstName` and `LastName` in Figure 2.1), or n-1 (e.g., `Street`, `City`, and `State` vs `Address`) mappings. For a comprehensive survey on matching approaches, we refer the readers to the work of Rahm et al. [119].

For example, in the case of the running example, COMA++ [44] gives us the result shown in Table 2.1 by running the default matching strategy. As can be seen, COMA++ chooses the correspondences with highest scores and beyond a predefined threshold (0.6) from the similarity matrix. Also, COMA++ provides the wrong correspondence `Source.Order.Address` vs `Target.Order.Address`. Instead, `Street`, `City`, and `State` of the source schema correspond to `Address` of the target schema.

| Mapping Element | Source | Target | Score |
|-----------------|----------|----------------------|-------|
| 1 | Order | Order | 0.748 |
| 2 | Id | OrderId | 0.684 |
| 3 | Quantity | Quantity | 0.849 |
| 4 | Price | Price | 0.855 |
| 5 | ProdName | ProductName | 0.736 |
| 6 | Address | Address | 0.608 |
| 7 | Name | FirstName & LastName | 0.687 |

Table 2.1: List of the semantic correspondences produced by COMA++ for the running example

Despite the development of numerous matching techniques, schema matching generally suffers from poor precision and recall as presented in [80] (e.g., 40% precision and 45% recall in ontology-matching tasks). As a result, human intervention is usually needed to validate and refine matching result. Spicy [51] proposes a new way to mitigate human intervention, namely *mapping verification*. This approach assumes that the instance of the target data source are available, in addition to the target schema (e.g., in the cases of Web data sources). Whenever a set of candidate

correspondences is produced via a schema matching module, the generated transformation result is compared against the available target instance. Based on such comparison, users are able to determine plausible correspondences produced by the schema matching module.

Apart from COMA++ [44], current matching systems are mainly not readily public for download and testing and matching results are only available in publications, such as Cupid [108] and Similarity Flooding [13]. As a result of that, a recent integration project, namely Open Information Integration (OpenII) [24, 30], has been developed to provide open integration components. One of them is the open-source schema matching component, namely Harmony.

2.4 Transformation Languages

Due to ubiquity of XML as a standard for heterogeneous data exchange [127], we mainly focus on XML-to-XML transformation languages in this section. Several languages have been proposed for transforming between XML documents (e.g., XML-QL [63], Quilt [56], XSLT and XQuery). Among these languages, XSLT [12], [17] and XQuery [16] are the most well-known and widely adopted ones, which are recommended by W3C. Two languages are equally capable as they are all largely based on XPath [11], [15]. It is worth noting that while XSLT is designed to transform documents, the main design purpose of XQuery is to query documents. However, the line between querying and transformation is increasingly blurred. There is an enormous overlap between features and capabilities of these two languages. Most industry mapping tools and research prototypes support generating both XSLT and XQuery. Section 2.4.1 provides an overview of XPath, which is the foundation for both XSLT and XQuery. Sections 2.4.2 and 2.4.3 describe the key features of XSLT and XQuery, respectively. Finally, Section 2.4.4 discusses differences between XSLT and XQuery as well as points out the limitations of the two languages.

2.4.1 XPath

XPath is an expression language that allows users to navigate through elements and attributes in an XML document. An expression takes one or more input and returns a value as output. The XPath value that results from evaluating an expression is known as a sequence, which is a collection of items and may contain atomic values as well as nodes. The syntax is a mix of basic programming languages (e.g., `$Price*5`) and Unix-like path expressions (e.g., `/Source/Order/Id`). The syntax of XPath is primarily similar to Uniform Resource Identifiers (URI), but navigation is via nodes in a XML tree rather than a physical file structure. Paths are interpreted regarding the current context node (the node is being processed). For example, simple location expressions are: (i) `/`, which selects the root node; (ii) `.`, which is shorthand for the current context node; (iii) `..`, which selects the parent node of the context node.

XPath is declarative, rather than procedural: It relies on patterns to describe types of nodes to look for, including *axis* (specify a directional relationship in the node tree such as `parent::`), *test* (define the nodes to select using the node name or type, such as `"*"`, `"@"`, `"node()"`), and *predicate* (use to filter a selection by using conditions on nodes, such as `//Order[Name='Ford Prefect']`).

Besides, XPath provides a set of useful functions on strings (e.g., *concat*, *contains*, *substring-before*), numbers (e.g., *avg*, *sum*, *max*), and date/time (e.g., *current-date*, *current-time*) that allow manipulating various node values in an XML document.

For example, given the XML document in Listing 2.1, some valid XPath expressions are:

- `/Source/Order/Name` using the absolute XPath expression selects all the `Name` elements which are children of the `Order` elements; the `Order` elements, in turn, are children of the root element `Source`.
- The expression `//Order[Name="Ford Prefect"]` using the relative path with a predicate returns all the `Order` elements and each of which has the child element `Name` equalling "Ford Prefect".

- Function *concat* can be used to merge values of three fields **Street**, **City**, and **State** with delimiters whitespace “ ”, and returns a string value.

2.4.2 XSLT

An XSLT program (i.e., stylesheet) is a set of *template rules* and each of which has two parts including a matching pattern that is matched against nodes of a source XML document and template content that can be instantiated to form part of the result tree. A template declaration is of the form:

```
<xsl:template match="match expression">
    content
</xsl:template>
```

where *match expression* is an XPath expression selecting source nodes that the template applies to, and *content* is a sequence of XSLT elements containing operations for transforming selected nodes as well as inserting nodes and textual content into the result tree. Some common operations are writing textual content, copying nodes and values from the source document, and generating new elements and attributes.

A template is applied by means of `xsl:apply-templates` and `xsl:call-template` (for named templates) elements inside content part of `xsl:template` elements. The execution of an XSLT style sheet on an XML instance starts with applying the template matching the outermost element of the instance. Afterward, the execution process is directed by elements `xsl:apply-templates` and `xsl:call-templates`. Some instructions for generating output content are `<xsl:copy-of select='expression'>` and `<xsl:value-of select='expression'>`. Attributes values can be generated by `<element-name attribute='expression'>`.

Other advanced feature of XSLT 2.0 [31] are branching elements including `<xsl:if>`, `<xsl:choose>`, `<xsl:for-each>`, `<xsl:variable>` (for variables declaration), and `<xsl:sort>` (for sorting). XSLT 2.0 also supports grouping via element `<xsl:for-each-group>` with attribute `group-by` for selecting grouping attributes. With this element, functions `current-group()` and `current-grouping-key()` are

used to work with grouping data.

For example, Listing 2.2 shows XSLT code for transforming the source instance in Listing 2.1 to the target format conforming to the target schema according to the transformation scenario in Figure 2.1.

Listing 2.2: Part of XSLT Code for implementing the running example

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/Source">
    <Target>
      <xsl:apply-templates select="Order"/>
    </Target>
  </xsl:template>
  <xsl:template match="Order">
    <Order>
      <xsl:apply-templates select="Id" />
      <xsl:apply-templates select="Name" />
      <xsl:apply-templates select="Address" />
      <xsl:apply-templates select="ProdName"/>
      <xsl:apply-templates select="Quantity"/>
      <xsl:apply-templates select="Price"/>
    </Contact>
  </xsl:template>
  <xsl:template match="Id">
    <OrderId>
      <xsl:value-of select="."/>
    </OrderId>
  </xsl:template>
  <xsl:template match="Name">
    <FirstName>
      <xsl:value-of select="substring-before(., ' ')" />
    </FirstName>
    <LastName>
      <xsl:value-of select="substring-after(., ' ')" />
    </LastName>
  </xsl:template>
  <xsl:template match="Address">
    <Address>
      <xsl:apply-templates select="Street" />
      <xsl:text> </xsl:text>
      <xsl:apply-templates select="City" />
      <xsl:text> </xsl:text>
      <xsl:apply-templates select="State" />
    </Address>
```

```
</xsl:template>
...
</xsl:stylesheet>
```

2.4.3 XQuery

As mentioned earlier, XQuery is primarily designed as a query language for data stored in XML documents. The simplest kind of query is to select elements or attributes from an input document known as path expressions [138]. Path expressions consist of a series of steps, separated by slashes, and each of which traverses the elements and attributes in the XML documents. Syntax of a path expression is identical to the one of XPath. Path expressions are simple and easy-to-learn, but their limitation is that they can only return elements and attributes appearing in input documents. FLOWR (“for, let, where, order by, return”) expressions overcome this limitation by allowing users to manipulate, transform, sort, group, and join results returned by path expressions: **for** is used to set up an iteration through a set of nodes; **where** is used to set up conditions for selecting nodes; **order by** sort the query results according to source atomic nodes in a certain order; **return** indicates results that should be returned; **let** is used to set the value of a variable, and unlike **for**, it does not set up an iteration.

XQuery also provides more than 100 functions that can be used to manipulate strings (e.g., **substring**, **concat**, **string-len**) and dates (e.g., **dateTime**, **years-from-duration**, **hours-from-time**), perform mathematical operations (e.g., **number**, **round**, **floor**), and other useful scenarios. In addition to these built-in functions, users are able to define their own functions.

Here are some examples of valid XQuery snippet:

- `doc('Orders.xml')//Order` returns all `Order` elements of the `Order.xml` document.
- Listing 2.3 performs the transformation scenario depicted in Figure 2.1.
- Listing 2.4 shows an example of user-defined function for reformatting date

from Australian format (dd/mm/yyyy) to US format (mm/dd/yyyy).

Listing 2.3: Part of XQuery code for implementing the running example

```
<Target>
{
  for $Order in doc('Orders.xml')/Source/Order
  return
  <Order>
    <OrderId>{$Order/Id/text()}</OrderId>
    <FirstName>
      {tokenize($Order/Name/text(),' ')[1]}
    </FirstName>
    <LastName>
      {tokenize($Order/Name/text(),' ')[2]}
    </LastName>
    <Address>
      {concatenate($Order/Address/Street,' ', $Order/Address/City,
        ' ', $Order/Address/State)}
    </Address>
    <Item>
      <ProductName>{$Order/ProdName/text()}</ProductName>
      <Quantity>{$Order/Quantity/text()}</Quantity>
      <Price>{$Order/Price/text()}</Price>
    </Item>
  </Order>
}
</Target>
```

Listing 2.4: An XQuery User-defined Function

```
declare function func:format-date($dt as xs:string) as xs:string
{
  let $refDateStr := string($dt)
  let $year := substring($refDateStr,1,4)
  let $month:= substring($refDateStr,6,2)
  let $day := substring($refDateStr,9,2)
  return concat($month,'/',$day,'/',$year)
};
```

2.4.4 Discussion

XQuery is designed for querying XML documents, but also often used for transforming between XML documents that directly competes with XSLT. One is not

powerful than other (both of them are *Turing-complete* [98]); the choice of a language is mainly a matter of personal preference. XQuery code is less verbose than XSLT and is more familiar with SQL users. XSLT 2.0 and XQuery 1.0 share many components:

- Both languages share the same data model including concepts of sequences, atomic values, nodes, and items.
- While XQuery 1.0 is a superset of XPath 2.0, XSLT 2.0 makes use of XPath 2.0 expressions from matching templates to copy nodes from input documents.
- Most built-in functions of both languages are identical. All of operators, such as comparison and arithmetic operators, yield the same values in both languages.

There are also some subtle differences between two of them due to design purposes [110]:

- While XSLT focuses on processing most of document which is reasonable for publishing, XQuery assumes users want to zoom in a few sections of document.
- XSLT assumes the process information is mostly textual, so XSLT is not a strongly type language. XQuery, in contrast, is a strongly typed language.
- XSLT assumes the generated document is presented in a markup language so XSLT is presented in an XML vocabulary while the syntax of XQuery is not XML-based.

Although the two languages are quite powerful, it is very tedious and error-prone to express transformations by hand in these languages (as shown in Listings 2.2 and 2.3. Code is typically verbose that makes programs difficult to maintain and debug. More importantly, it requires deep programming expertise which is far beyond the level of knowledge workers. Mapping technology is, therefore, proposed to help automate transformation tasks. We will discuss mapping technology in details in the next section.

2.5 Mapping systems

Mapping tools are designed to help users (semi)-automatically specify transformations using graphical user interfaces (GUI). Then, mapping tools translate a transformation specification into executable language (e.g., XSLT or XQuery), which can later be deployed to an execution engine to perform transformation over input data instances. Different tools have different levels of language support. For example, Stylus Studio supports many more XSLT functions than Clio does [38].

Most mapping tools, either research prototypes (Clio [85], Clip [117], +Spicy[112]) or industry tools (e.g., Atova MapForce [1], IBM Rational Data Architect [7], Stylus Studio [9], Microsoft BizTalk Mapper [3]) are basically relationship-based mapping systems [127]. More specifically, the visual interface of such mapping system displays the source schema and the target schema on the left side and the right side of the screen, respectively. Relationships between the source and target schemas are specified via connecting lines from the source elements to the target elements. In complex transformation scenarios, lines may be annotated with one or more functions/conditions of a built-in library for performing various transformation patterns, such as sorting, grouping, and filtering values.

For instance, a screenshot of mapping tool Altova MapForce for implementing the running example is shown in Figure 2.3. The mapping interface (shown in the **Mapping** tab) can be described as follows:

- While function *concat* of XPath is used to merge values of three source attributes **Street**, **City**, and **State** into values of target attribute **Address**, function *substring-before* is used to split values of attribute **Name** into values of attribute **FirstName**.
- Copying values of attributes **Id**, **ProdName**, and **Quantity** to values of attributes **OrderId**, **ProductName**, and **Quantity** are performed by connecting lines without requiring any functions.
- Function *multiply* is used to convert values of source attribute **Price** to formats required by values of target attribute **Price**.

- Source structural element **Order** is connected to target structural element **Order** to establish a structural mapping.

As can be seen, a library of built-in functions is located on the **Libraries** pane; the **Message** pane displays validation warnings or error warnings; the **Overview** pane displays the current mapping area as a red rectangle, which is helpful when a mapping is complex; the **Output** tab can be clicked to see the transformation result; the **XQuery** tab can be clicked to see generated XQuery code; the **Database Query** tab allows directly querying any major database.

Note that the structural mapping connected between two source and target element **Order** is needed to generate a valid target instance (with two orders 42 and 525). If this structural mapping is missing, an invalid target instance is produced, in which each element of the target schema appears twice. Meanwhile, in the case of Clio [85] and Stylus Studio [9], users do not have to establish this structural mapping to generate the valid target instance. This is because there is no standard for the interpretation of visual metaphors and constructs [37].

In the following, we focus on describing two recent research prototypes, namely Clio [85, 115] and Clip [117, 118], along with their theoretical foundations.

2.5.1 Clio

Clio [115, 79, 85] is the first mapping system that formalizes the relationship between the correspondences of the source and target attributes and the constraints of the source and target schemas. It uses *tuple-generating dependencies (tgds)* [71] to describe this formalism. More specifically, in the first phase (namely semantic translation), a set of inter-schema correspondences is converted into a set of mappings that capture the design choices made between the source and target schemas. The second phase (namely data translation) translates these mappings into executable queries that translating the source instance into the target format satisfying the constraints and structure of the target schema. The mapping generation algorithm of Clio can be briefly described as follows.

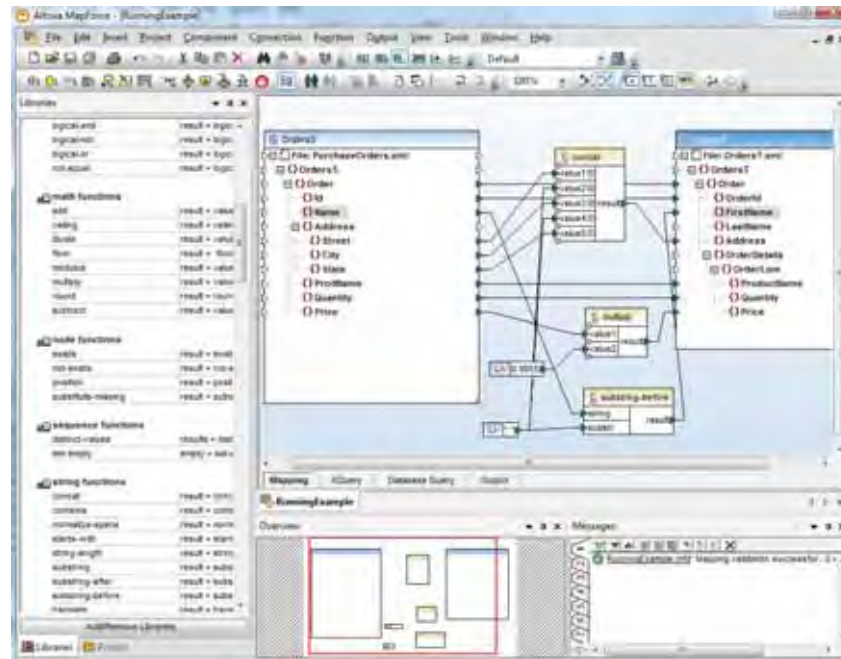


Figure 2.3: Mapping interface of commercial tool Altova MapForce for implementing the running example

First, it is needed to identify *logical relations* of the source and target schemas. Logical relations are maximal tableaux. A tableau is a set of semantically related schema elements: elements are related when they are siblings under the same set element or when set elements are related via a parent-child relationship. In order to generate logical relations, the algorithm finds *primary paths* in each schema. These are linear tableaux obtained by the enumeration of all paths from roots to any intermediate node of set type in the schema. Then logical relations are retrieved by chasing intra-schema constraints (e.g., key/foreign keys) against primary paths.

After the source and target logical relations are computed, a 2-dimensional matrix is created where one dimension is source logical relations and the other is target logical relations. Each entry in this matrix relates a source logical relation with a target logical relation and is called a *mapping skeleton*. A mapping skeleton is called *active* if it contains at least one established value mapping. Each active skeleton, that is not *implied* or *subsumed* by others, emits a *logical mapping* [115]. Each logical mapping is characterized by a tuple generating dependency (tgd) [71]. Such tgds represent different ways to cover the correspondences and generate tuples in

the target.

After that, generated logical mappings are possibly related in the sense that they share part of source and target expressions. In such cases, it is possible to nest logical mappings inside each other to reduce the overall number of mapping expressions which are called *nested mappings* [79]. As a result, the execution stage will generate much less redundancy in the target data.

For example, consider the mapping generation for the running example. We have the following logical relations:

- $S_1 = \{o \in \text{Source.Order}\}$
- $T_1 = \{o' \in \text{Target.Order}\}$
- $T_2 = \{o' \in \text{Target.Order}, l' \in o'.\text{Item}\}$

There are two active mapping skeletons, namely (S_1, T_1) and (S_1, T_2) , but (S_1, T_1) is implied by (S_1, T_2) . Hence, only one logical mapping is emitted by (S_1, T_2) . The corresponding tgds is:

```

 $\exists$  concatenate, left, right, len, search( $\forall o \in \text{Source.Order} \rightarrow \exists o' \in$ 
Target.Order,  $l' \in o'.\text{Item} \mid o'.\text{OrderId} = o.\text{Id}, l'.\text{Address} =$ 
concatenate( $o.\text{Street},'',o.\text{City},'',o.\text{State}$ ),  $l'.\text{FirstName} =$ 
left( $o.\text{Name},\text{search}('',o.\text{Name})$ ),  $l'.\text{LastName} =$ 
right( $o.\text{Name},\text{len}(o.\text{Name})-\text{search}('',o.\text{Name})$ ),  $l'.\text{Price} = o.\text{Price}/0.85$ 
+ 5,  $l'.\text{Quantity} = o.\text{Quantity}$ ,  $l'.\text{ProductName} = o.\text{ProdName}$ )

```

As can be seen, functions `concatenate`, `left`, `search`, `len`, and `right` (Excel-like functions) are included in the mapping. Additionally, Clio is also equipped with a matching component to reduce the burden of discovering semantic correspondences for users.

The code generation step produces XQuery code from generated tgds. The query generation algorithm of Clio [79] takes as input a nested mapping M and produces an XQuery FLWOR expression as output. Each sub-mapping of M is translated

into one nested FLWOR expression of F. F has the following structure: a “for” clause captures the iteration implied by every universally quantified variable of M ; a “where” clause captures the join and filtering predicates; a “return” clause constructs the XML items for the target schema elements mentioned in the existentially quantified part of the mapping; elements bound to some of the variables defined in the for clauses are copied to the proper positions according to the values mappings expressed in the mapping M . In turn, the sub-mappings of M recursively replicate this structure. The target data is then nested by default according to Partitioned Normal Form (PNF) [33]. Moreover, target required attributes with no correspondences to the source elements are created using Skolem functions. For example, XQuery code generated from the above tgd-based mapping description of the running example is as follows (without PNF):

```
<Target>
{
  foreach $o in Source/Order
    return
      <Order>
        <OrderId>$o/Id/text()</OrderId>
        <FirstName>left($o/Name/text(),search(' ', $o/Name/text()))
        </FirstName>
        <LastName>right($o/Name/text(),string-length($o/Name/text())
- search(' ', $o/Name/text()))</LastName>
        <Address>concat($o/Street/text(),',', $o/City/text(),',',
, $o/State/text())</Address>
        <Item>
          <ProductName>$o/ProdName/text()</ProductName>
          <Quantity>$o/Quantity/text()</Quantity>
          <Price>$o/Price/text()/0.85 + 5</Price>
        </Item>
      </Order>
}
</Target>
```

There has been numerous extensions to Clio to implement the model management vision pioneered by Bernstein et al. [47, 48]. So far, two operators for mapping manipulation, namely *mapping composition* [74] and *mapping inversion* [70, 75], are supported by Clio.

Recent work on *schema covering*, the problem of identifying easily understandable objects for describing large and complex schemas, is a major step towards enabling transformation reuse for Clio [128]. It assumes that there is a repository of concepts and transformations among them. A concept represents a basic business object such as employee, product, article, and so on that is at higher-level abstraction than a schema and therefore is easier to understand for users. Given the repository, schema covering finds the relevant concepts and the mappings from the source and target schemas to those concepts. Then using existing mappings between concepts in the repository, transformation from the source to the target can be computed by composing such individual mappings.

2.5.2 Clip

Clip [118] extends Clio’s mapping generation algorithm which allows users to explicitly control *structural mappings*. In other words, besides value mappings (i.e., mappings between two attributes of the source and target schemas), it introduces structural mappings that connect between structural elements (i.e., *SetOf* elements) of the source and target schemas. As a result of that, it produces more accurate and expressive mappings, such as filtering, sorting, and explicit grouping. Clip allows users to explicitly build nesting mappings via structural mappings and *context propagations trees* (CPTs). A structural mapping may be associated with a *build node*. Build nodes can also be connected from one to another via context arcs to form CPTs. A CPT is indeed a nested mapping.

Additionally, Clip’s build nodes correspond to Clio’s mapping skeletons. For each build node, source side builders are matched against the computed source tableaux. If a build node appears in a CPT, source-side builder are matched against source tableaux. If no source tableau is found, a new tableau is created that will cover

source builders and added to the list of Clio’s tableaux. It is done similarly for the target side of each builder. At the end, the source and target tableaux that form the context of the build node are identified. Thus, this tableaux pair is the Clio mapping skeleton that matches the build node. Clip also extends the query generation algorithm of Clio to take into account minimum-cardinality assumption, explicit grouping, and aggregating.

For instance, in the case of the running example, suppose that the target instance is grouped by the grouping attributes $\{\text{Id}, \text{Name}, \text{Street}, \text{City}, \text{and State}\}$. The user needs to use function `groupby` and its input parameters are these grouping attributes. Clip generates the following second-order tgd to describe this mapping:

```

 $\exists$  groupby, concatenate, left, right, len, search( $\forall o \in \text{Source.Order}$ 
 $\rightarrow \exists o' \in \text{Target.Order}, l' \in o'.\text{Item} \mid o' =$ 
groupby( $o, o.\text{Id}, o.\text{Name}, o.\text{Street}, o.\text{City}, o.\text{State}$ ),  $o'.\text{OrderId} = o.\text{Id},$ 
 $l'.\text{Address} = \text{concatenate}(o.\text{Street}, '', o.\text{City}, '', o.\text{State}), l'.\text{FirstName}$ 
 $= \text{left}(o.\text{Name}, \text{search}('', o.\text{Name})), l'.\text{LastName} =$ 
 $\text{right}(o.\text{Name}, \text{len}(o.\text{Name}) - \text{search}('', o.\text{Name})), l'.\text{Price} = o.\text{Price}/0.85$ 
 $+ 5, l'.\text{Quantity} = o.\text{Quantity}, l'.\text{ProductName} = o.\text{ProdName}$ )

```

2.6 Data exchange

Data exchange generalizes the theoretical foundation of the mapping system Clio to develop more generic solutions for the data exchange setting. In general, given a source instance I , there may be possible multiple solutions for the target instance under a schema mapping $M = \{S, T, \Sigma_{st}, \Sigma_t\}$, instead of only one canonical universal solution as in the case of mapping tools. Here, S and T are the source and target schema respectively; Σ_{st} is a set of source-to-target dependencies; Σ_t is a set of target dependencies. The space of all solutions for I under M is denoted as $Sol(M, I)$. Suppose that Σ_t is the union of a weakly acyclic set of tgds with a set of equality-generating dependencies [46], chase I under $\Sigma_{st} \cup \Sigma_t$ produces a *universal solutions* in polynomial time [71] if a solution exists. In a nutshell, a universal solution

has no more or no less data than required for the data exchange problem, and is therefore preferable. In fact, Clio and other tgD-based mapping systems (e.g., Clip and +Spicy [111]) implement a naive chase compared with data exchange.

For example, consider a data exchange problem in which the source schema has two relation symbols P and Q , each of which with attributes A and B , while the target schema has one relation symbol T also with attributes A and B . Suppose that $\Sigma_t = \emptyset$. The source-to-target dependencies and the source instance are:

- $P(a, b) \rightarrow \exists X T(a, X)$
- $Q(a, b) \rightarrow \exists Y T(Y, b)$
- $I = \{P(a_0, b'_0), Q(a''_0, b_0)\}$

Note that the dependencies in Σ_{st} do not completely specify the target instance. As a result, there may be solutions for I . Two of them are:

- $J_0 = \{T(a_0, X_0), T(Y_0, b_0)\}$ where X_0 and Y_0 represent unknown values called *labelled nulls*.
- $J_1 = \{T(a_0, b_0)\}$

We can see that J_1 does not use labelled nulls and are replaced with source values. Solution J_1 seems to be less general than J_0 since it makes assumption that all two tuples required by the dependencies are equal to the tuple $\{T(a_0, b_0)\}$. In fact, J_0 is a universal solution that contains no more or no less than what the specification requires.

As we mentioned above, another problem with current mapping systems is that they generate a lot of redundancy for the target data. Although the work [79] propose nested mappings to remove redundancy of Clio's algorithm, its solution is not generic enough to apply to a large number of cases. The notion of *core solution* is introduced in [73] to deal with redundancy in a generic way. In a nutshell, a core solution is the optimal solution for data exchange. +Spicy [111] is the first mapping system that integrates and implements core solutions by rewriting tgD expressions of Clio.

2.7 String-based data transformation

There has been some work specifically focusing on string-based transformation, which converts one or more strings to another. Tope [132] enables users to define string-based types at higher level of abstraction in comparison with types of traditional programming languages (e.g., strings, integer, float), such as address, phone number, person name, email, and currency. This is done via an easy-to-use and form-based editor, namely Toped++ [130]. Context-free grammar (CFG) is then generated accordingly for each defined type [131]. Each type (Tope) has multiple formats. For example, Tope person name has at least two formats, namely “FirstName LastName” and “LastName, FirstName”.

A format of a Tope has its constituent parts. For example, a US phone number has three parts: area code, exchange, and local number. The user can add remove, and reorder parts, and can specify constraint-like facts on parts. A US phone number format can be described by the following CFG:

- `number` \rightarrow `area - exch - local`
- `area` \rightarrow `d d d {c=60}`
- `exch` \rightarrow `d d d`
- `local` \rightarrow `d d d d`
- `d` \rightarrow `0 — 1 — 2 — 3 — 4 — 5 — 6 — 7 — 8 — 9`

It means that area code, exchange, and local number contain three, three, and four digits respectively. Note that `c` stands for a “confidence” in the range of 0 and 100; 100 indicates that the constraint is always true; 0 indicates that it is never true. In this example, constraint `c=60` is “often” true.

Based on generated CFGs of formats, a parser can be developed to validate an input string against a given format. The validation result returns a number between 0 and 1 to indicate the parser’s confidence in each string’s validity. The parser rejects any input with a confidence of 0 and accepts any input with a confidence of 1. If

the confidence is between 0 and 1, the parser consults the user’s confirmation rather than rejecting the input. This is less constraining than using regular expressions.

A Tope has its own functions for automatically transforming one format to another [130]. For example, it is able to automatically transform “Bill Gates” to “Gates, Bill” using a reformatting function of Tope person name.

Potter’s Wheel [121] is an interactive data cleaning up system that tightly couple transformation and discrepancy detection inside a spreadsheet-like interface which is very familiar to end-users. Data cleaning basically has three components: auditing data to find the discrepancies, choosing transformations to them, and applying them on the data set. Users gradually add or undo transformations in a intuitive and graphical manner through the spreadsheet-like interface. The result of a transformation is shown instantly on screen and it can be undone easily if its effect is undesirable.

By proposing a number of common form-based transformation operators (e.g., `format`, `add`, `drop`, `copy`, `merge`, `split`), Potter’s Wheel enables users to perform various transformations graphically instead of writing complex programs; writing code for transformation is time-consuming and error-prone, and distracts users from the main job of detecting discrepancies and choosing transformations to fix them. Instead, transformations should be specified via intuitively GUI-based operations, or at worst by outlining the desired effect on example values, which will be discussed in Section 2.8. For example, operator `split` allows the user to graphically split string “Gates, Bill” into two strings “Bill” and “Gates” according to splitters “ ” and “,”.

Wrangler [97] extends Potter’s Wheel with additional operators for common cleaning tasks such as positional operators, aggregation, semantic roles, complex reshaping operators, and conditional mapping operators (e.g., update country to “US” where state=“California”). As users select data, Wrangler suggests applicable transformations based on the current context of interaction. Transformation steps of users are recorded in a script to facilitate reuse and provide documentation of data provenance. Wrangler’s interactive history viewer supports review, refinement, and annotation of these scripts.



Figure 2.4: Simultaneous editing interface of PotLuck

PotLuck is a Web-based integration tool designed for end-users to mashup data [89]. It let casual users pool together data from several sources, supports drag and drop for merging fields, integrates and extends the facet browsing paradigm for focusing on subsets of data to align, and applies simultaneously editing for cleaning up data syntactically. Regarding simultaneous editing, PotLuck groups field values into columns by structural similarity (e.g., the phone numbers in a column all have area code 212). These are used to visually separate out values of different forms and let the user edit different forms differently. The user can click on any field value to get focus, start editing, and all editing changes are applied to other values in the same column in a similar fashion. An example of simultaneous editing is shown in Figure 2.4.

These techniques of PotLuck are further developed and improved in GridWorks [22], a tool of Metaweb for cleaning and analyzing data which is uploaded by users or come from Freebase [21]. This tool is recently renamed Google Refine [90].

2.8 Transformation by examples

An effective way to reduce hassle for casual users in transformation specification is to learn transformation rules from examples provided by the user, and then apply those rules to the transformation task at hand. It is similar to the principle of *programming by example* (or programming by demonstration) systems [62]. Specifically, the user demonstrates a set of actions on an example, these systems tries to infer application

(transformation) logic via learning (generalization) from the example. The logic can then be applied to other similar tasks.

CopyCat [18] is a form of programming by examples: (i) The user demonstrates the actions to be performed to integrate data (copying data from source applications to CopyCat); (ii) The system learns to generalize users' actions; (iii) The system immediately shows the effects of applying these generalizations in the form of auto-complete suggestions, and solicit feedback on these suggestions. For example, suppose one integration task is to take a list of shelters from a television Website, combine it with the shelters' contact stored in a spreadsheet. The user can load the page of shelters into her Web browser. She then selects and copies the first item, then pastes it into the CopyCat workspace. CopyCat tries to generalize the user's action and copies other shelters from the same page and adding new rows to the workspace. The user might accept or reject these new rows.

As an example of transformation by examples, Potter's Wheel [121] enables users to specify most splits by performing them on examples. As a result, the user is able to parse and split values without specifying complex regular expressions or writing programs. More specifically, the user selects a few examples values v_1, v_2, \dots, v_n and in a graphical and direct-manipulation way shows how these are to be split into components $(w_{11} \ w_{12} \dots w_{1m}), \dots, (w_{n1}, \dots, w_{nm})$. Potter's Wheel then infers a structure for each of m new columns and uses these structures to split the rest of values.

Arvind et al. proposes methods [42] to learn string transformations from examples in the context of record matching [67]. In reality, "Robert" and "Bob" may refer to the same first name, but are syntactically different. Consequently, traditional string similarity functions are not flexible enough to account for such synonyms. Work [41] proposes a framework for addressing such representational variations by incorporating a priori knowledge of such variations into record matching process. It uses string transformations to refer to such alternate representations (e.g., **Robert** \rightarrow **Bob**). Unfortunately, for a real-world matching task, thousands of string transformations could be too time-consuming and labour-intensive and it is a challenging task for a programmer to compile such set of relevant transformations. Thus, work [42] considers the approach of learning transformations from user-provided examples of

matching strings. The key idea is that by drawing examples from the data sets being matched, it is able to identify transformations that are appropriate for the matching task at hand.

2.9 Summary

In this chapter, we have provided a background and briefly described main approaches in data transformations. In summary, there are two main limitations with respect to current approaches:

- Data transformation solutions mainly target professional developers, rather than knowledge workers with no programming background, who wish to apply data transformation to their daily jobs. Although visual mapping tools have been designed to help users specify mappings via GUI, instead of writing low-level code, they are still unintuitive and cumbersome in terms of mapping interface and transformation functions.
- Previous transformation efforts are not leveraged effectively in order to reduce effort duplication. Schema covering is first proposed in [128] as a step towards transformation reuse. However, this work assumes the existence of a repository of concepts and mappings among those concepts. Building such repository is challenging and time-consuming. Moreover, existing mappings between these concepts must be composed, which is also a complicated task.

Given such limitations, we believe that there are two issues that mapping technology should facilitate and deserve further work: (i) an intuitive and familiar environment for knowledge workers to specify transformations; (ii) a reuse mechanism for leveraging past transformations effectively and efficiently. In this dissertation, we will try to address these issues in the context of spreadsheet-based data transformation.

Chapter 3

Spreadsheet-based data transformation language

In this chapter, we investigate the problem of developing a spreadsheet-based data transformation language. 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.

3.1 Introduction

As mentioned earlier in Section 2.1, given the fact that a significant amount of the world’s data is maintained in spreadsheets, it is now increasingly necessary for using data stored in spreadsheets to interact with external applications and

services [114]. One of the key problems is transforming spreadsheet data to the structured formats required by these services and applications. This problem is different from traditional data transformation [71] because of: (i) characteristics of the source data model (spreadsheets); (ii) users that we aim at, who are familiar with the spreadsheet programming paradigm.

In this chapter, we consider the problem of developing a spreadsheet-based data transformation language. 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 3.1(a) contains data arranged in a table while the schema used by a bar chart (Figure 3.1(d)) consists of a list of labels, each comprising a list of bars.

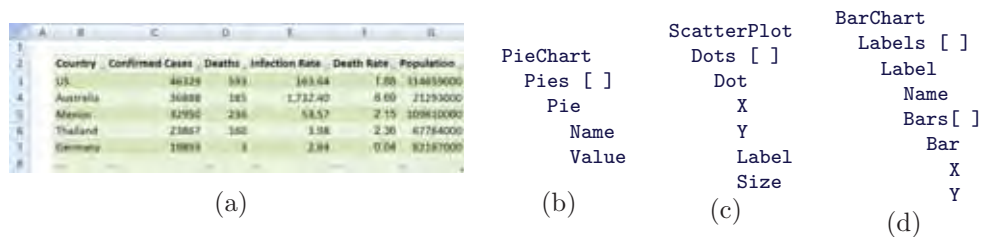


Figure 3.1: (a) The Swine Flu data set; (b) *Pie Chart* schema; (c) *Scatter Plot* schema; (d) *Bar Chart* schema

Mainstream 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 [72]. 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 [136, 8, 5] 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 [104, 60], 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 [79], Clip [118], +Spicy [111], and Altova MapForce [2]. However, users must learn a new language, e.g., by creating correspondences between the source and target elements and annotating those correspondences with one or more unfamiliar functions (e.g., functions of XSLT/XQuery, Java, C#) in the case of mapping tools [127]. This flowchart-like mapping interface is typically cluttered when schemas are large and mappings are complex [126]. On the contrary, spreadsheet users are familiar with formulas and an incremental approach to building applications with instant feedback [96, 106, 135, 139].

The second approach, namely *column-based* [123, 53], enables users to specify simple mappings between spreadsheet columns and target attributes (atomic elements) via drag-and-drop operations. 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 3.1(c)) onto source column **Infection Rate** (Figure 3.1(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 [136, 5]. 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 3.1(a)) using a pie chart (Figure 3.1(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 3.1; (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.

To address these issues, we propose a novel approach, namely TranSheet, which enables users to perform mappings via a familiar and expressive spreadsheet-like formula language. The syntax and semantics of the language are mainly based on MS Excel due to its ubiquity, but our design is also applicable to other spreadsheet systems (e.g., OpenOffice Calc, Google Spreadsheets, Gnumeric, and Apple Numbers). 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 popular transformation patterns that are relevant to spreadsheet-based transformation are supported in the language via spreadsheet formulas and functions. These patterns are the result of a careful analysis of commonly needed mapping scenarios provided by transformation languages (e.g., XSLT/XQuery), mapping tools [37], and spreadsheet corpuses [77]. This enables the language to avoid cluttering the spreadsheet with transformations and it turns out to be helpful when multiple schemas are targeted (Section 3.3).
- Target schema restructuring is proposed to allow users to resolve the mismatches between the source spreadsheet and the target schema. It is a common occurrence that the target schema, which is defined externally, does 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 to match the spreadsheet organization. By rearranging the schema view, users do not modify the underlying target schema; they merely specify how mapping formulas should be interpreted (Section 3.4).

- 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 spreadsheet templates are exploited to generalize mappings. Consequently, TranSheet can transform a large number of naturally occurring spreadsheets, that cannot be effectively handled by the alternative approaches (Section 3.5).
- We use *tuple generating dependencies* (*tgds*) [72, 143], a widely used schema mapping formalism, to describe the semantics of TranSheet. We introduce a collection of new functions to *tgds* expressions. We then extend a previous query generation algorithm [79, 118] to generate executable queries for these functions (Section 3.6).
- 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 data transformation scenarios (Section 3.8).

Note that within this chapter, we mainly focus the foundations and semantics of the language, rather than its usability aspects. The usability of the language is addressed in Chapter 4.

The rest of this chapter is structured as follows. Section 3.2 presents the data models of spreadsheets and target schemas. Next, Section 3.3 describes in details the formula mapping language. Section 3.4 introduces schema restructuring details that help users work around structural mismatches between the source spreadsheet and the target schema. Section 3.5 then provides the generalization of mapping formulas. Section 3.6 formally describes semantics of the language and how executable code is generated for transformation execution. Section 3.7 presents the implementation

details of TranSheet. Section 3.8 describe experiments in two real-world applications. We discuss related work in Section 3.9 and conclude in Section 3.10.

3.2 Data Model

We present the spreadsheet data model in Section 3.2.1 and then target data model in Section 3.2.2.

3.2.1 Spreadsheet data model

Unlike traditional database, a spreadsheet contains data in a poorly structured way. Spreadsheets are collections of cells organized in a grid or large table. Each cell is either empty or contains an atomic value. In that respect, spreadsheets are, at least, more structured than, say, free text.

Very often, a spreadsheet conveys much additional structuring information through the spatial organization of cells, i.e., through the way users lay out data on the grid. For example, a spreadsheet may contain data organized into tables where each column corresponds to an attribute. Unfortunately, this structuring remains implicit: It is known to the user who organized the data layout, it may be understandable to another human looking at the spreadsheet, but it cannot readily be used for mapping purposes since it is not available in an explicit and formal representation.

Note that some explicit and semi-formal structuring is also possible in modern spreadsheet environments. For example, MS Excel allows creating tables (*list* is the name used in Excel terminology) to distinguish header cells from content cells. Availability of such structure would ease the mapping problem. However, we will discuss here the most general settings, in which no such structures are available.

A spreadsheet S (where S , from now on, will stand indifferently for Source or Spreadsheet) is modelled as a two-dimensional matrix of cells.

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 = $\{ \langle x, y \rangle \mid 4 \leq x \leq 5, 3 \leq y \leq 13 \}$. A range $\langle x_1, y_1 \rangle : \langle x_2, y_2 \rangle$ always verifies $1 \leq x_1 \leq x_2$ and $1 \leq y_1 \leq y_2$. The spreadsheet environment also allows for the definition of ranges of non-contiguous cells. These are denoted by separating each cell or range of cells with a comma.

Each cell has an atomic value with associated type τ_a : $\tau_a ::= \{ \text{empty}, \text{int}, \text{float}, \text{string}, \text{datetime} \}$. The type *empty* represents the special case of an empty cell.

The spreadsheet data model has very special characteristics 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). These are significantly different from the data model of XML or the relational data model, where the structuring can be referred to in terms of the symbols (e.g. XML elements, table names and attributes). These characteristics will be later exploited in mapping generalization.

3.2.2 Target Data Model

In this chapter, we focus on transformation of spreadsheet data to XML, which is a popular standard for heterogeneous data exchange [127]. For this purpose, we use a slight variation of nested relational model proposed in [79, 116]. 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. Note that it is straightforward to convert XML to JSON [29], which is a standard format for Web mashups.

We model a schema as a set of typed labels (i.e., elements). The set τ_t of label types is defined as follows: $\tau_t ::= \tau_a \mid \text{SetOf}[l_i : \tau_t^i] \mid \text{Rcd}(l_1 : \tau_t^1, \dots, l_n : \tau_t^n) \mid \text{Choice}[l_1 : \tau_t^1, \dots, l_n : \tau_t^n]$. In this notation, l_i s are label names and τ_t^i s 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, *Choice* represents choice constructs and *Rcd* represents tuples (i.e., collection of ordered label-value pairs). For example, in Figure 3.2(b), we have:

- Label **Orders** consists of a set of labels **Order**.
- Label **Order** consists of a tuple {**Id**, **ShipTo**, **OrderDetails**}; label **Id** has type integer.
- Label **ShipTo** consists of three atomic labels **FirstName**, **LastName**, and **Address**; three labels **FirstName**, **LastName**, and **Address** have type string.
- Label **OrderDetails** consists of three atomic labels **Quantity**, **ProdName**, and **Price**. Label **Quantity** has type integer; label **ProdName** has type string; label **Price** has type float.

For presentation in the user interface, we use an equivalent notation illustrated in Figure 3.2(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 chapter, 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). If there is a transtyping violation when assigning a value of the spreadsheet to a given label, instant feedback is provided to users.

3.3 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). A mapping formula has the form $l = f$, where l is a target label and f is a spreadsheet-like formula.

| Types | Transformation patterns |
|--------------------|---|
| Value mapping | <i>Copying, Reformatting, Constant Value Generation, Merging, Splitting</i> |
| Structural mapping | <i>Nesting, Filtering, Sorting, Grouping with aggregation, Join and Cartesian product, Union/Minus/Intersect, Branching</i> |

Table 3.1: Transformation patterns of TranSheet

In the following, we formally define the syntax of the language and we demonstrate via examples that all popular transformation patterns provided by transformation languages and mapping tools are supported in our language (shown in Table 3.1). We emphasize that these patterns are not intended to cover all transformations; instead, they are proposed to capture common transformation scenarios that are relevant for spreadsheet-based data transformation. This is done by a careful analysis of commonly needed mapping scenarios of spreadsheet corpuses [77], transformation languages (e.g., XSLT/XQuery), and mapping tools [37, 118].

More specifically, Section 3.3.1 defines the context-free grammar of language’s syntax. In Section 3.3.2, we present the transformation patterns of value mappings. Next, Section 3.3.3 describes the transformation patterns of structural mappings. User-defined function support is discussed in Section 3.3.4. For each pattern, we define the semantics of its syntactical construct and illustrate via at least one example.

3.3.1 Formal definitions of language’s constructs

Formulas may refer to a cell or an expression of multiple cell references and other formulas. A formula expression is a composition of cell references, cell ranges, set operators, filter predicates, aggregations, sort operator, join operator, branching operator, and atomic value manipulations; each of which is optional. The composition of formula expressions must adhere to the following context-free grammar:

```

formula    => formula + term | formula - term | term
term       => term * factor | term / factor | factor
factor     => cellref

```

```

=> |cellrange
=> |number
=> |aggregations
=> |setopr
=> |joinopr
=> |branchopr
=> |avm
aggregations => aggrname(cellrange)
aggrname    => min|max|sum|
            => avg|count
setopr      => setoprname(setargs)
setoprname  => union
            => |intersect
            => |minus
setargs     => cellrange
            => |cellrange,cellrange
joinopr     => joinoprname(cellrange,cellrange,joincondition)
joinoprname => join
joincondition => cellrange = cellrange
branchopr   => branchoprname(filterexp,attrvalue,attrvalue)
branchoprname => if
avm         => cellref|cellrange|attrvalue|avmfunction
avmfunction => upper|lower|trim|...
            => |abs|ceiling|round|...
            => |date|time|hour|...
            => |concatenate
            => |left|search|right|len
cellref     => filterpred
cellrange   => filterpred:filterpred
            => |filterpred:filterpred[sortopr]
            => |filterpred:filterpred[groupopr]
            => |filterpred:filterpred[filterexp]
sortopr     => sortoprname({cellrange,sortvalue}+)
sortoprname => sort
sortvalue   => ascending|descending
groupopr    => groupoprname(groupattr,groupattr)
groupoprname => groupby
groupattr   => cellrange
            => |cellrange,cellrange
filterexp   => AND(filterexp,filterexp)
            => |OR(filterexpt, filterexp)
            => |NOT(filterexp)
            => |attrname = attrvalue
            => |attrname != attrvalue
            => |attrname > attrvalue
            => |attrname >= attrvalue
            => |attrname < attrvalue
            => |attrname <= attrvalue
filterpred  => cellid
cellid      => {$} {a-z}+ {$} {0-9}+

```


| | |
|-----------|----------------|
| attrname | => {a-z, 0-9}+ |
| attrvalue | => {a-z, 0-9}+ |
| | => {0-9}+ |
| number | => {0-9}+ |

In the next sections, we illustrate the semantics of each of these syntactical constructs of the language in details using examples.

3.3.2 Value mappings

Copying

This pattern simply copies a cell value to a target value of a label. It has the form $l = c$ where l is a target label and c is a single cell or a mono-dimensional range.

For example, in Figure 3.2(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). TransSheet, 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 3.2(b), the user inputs the mapping formulas `Quantity=B3:B17` (15 `Quantity` items) and `ProdName=C3:C5` (3

| | A | B | C | D |
|----|---------------------|----------------|------------------|------------|
| 1 | 0042 | | | |
| 2 | <i>Ford Prefect</i> | <i>Addison</i> | <i>Sydney</i> | <i>NSW</i> |
| 3 | | 150 | Beer | 75.64 |
| 4 | | 2 | Towel | 5.26 |
| 5 | | 1 | Babel Fish | 4.32 |
| 6 | | | | |
| 7 | 0525 | | | |
| 8 | <i>Arthur Dent</i> | <i>Evans</i> | <i>Melbourne</i> | <i>VIC</i> |
| 9 | | 1 | Towel | 2.75 |
| 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(D3:D5,0) {76, 5, ...}

```

(b) Schema view with mapping specifications for order 0042

Figure 3.2: A spreadsheet and its mapping specification

`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 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 whose children labels have definite values would be generated. Note those 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, it is needed to copy a constant to a target value where the constant value is independent of the source. This pattern has the form `l=const` where `l` is a target label and `const` is a constant.

For example, in Figure 3.2(b) 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. It has the form `l=f(c1,...cn)` where `l` is an atomic label, `f` is a function, and `ci` is a cell or a mono-dimensional range.

For example, in Figure 3.2(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. It has the form $l = \text{concatenate}(c_1, \text{const}_1, \dots, \text{const}_{n-1}, c_n)$ where l is a target label, c_i is a cell or a mono-dimensional range, and const_i is a constant.

For example, in Figure 3.2(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. It has the form $c = \text{concatenate}(l_1, \text{const}_1, \dots, \text{const}_{n-1}, l_n)$ where l_i is a target atomic labels; c is a cell or a mono-dimensional range; and const_i is a constant delimiter between l_i and l_{i+1} .

For example, mapping formulas `FirstName=left(A2, search(“”, A2))` and `LastName=right(A2, len(A2)-search(“”, A2))` in Figure 3.2(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*.

3.3.3 Structural mapping

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

| | A | B | C | D | E |
|---|----------------|------------------|-----------------|-----------------|-----------------|
| 1 | <i>OrderId</i> | <i>FirstName</i> | <i>LastName</i> | <i>ProdName</i> | <i>Quantity</i> |
| 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 =C2:C50 {Ford, Arthur, ...}
    LastName =B2:B50 {Prefect, Dent, ...}
  OrderDetails [ ]
    OrderLine
      ProdName {{Beer, Towel, ...}, ...}
      Quantity {{150, 2, ...}, {1, 20, ...}, ...}

```

(b) Mapping specification

Figure 3.3: A tabular representation of orders and its corresponding mapping

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 3.2 is interpreted in terms of lower level mapping formulas in the following two steps:

$$\begin{aligned}
 &\text{OrderDetails=B3:C5} \\
 &\quad \Downarrow \\
 &\text{OrderLine=B}\langle i \rangle\text{:C}\langle i \rangle, \quad 3 \leq i \leq 5 \\
 &\quad \Downarrow \\
 &\text{Quantity=B}\langle i \rangle, \text{ProdName=C}\langle i \rangle, \quad 3 \leq i \leq 5
 \end{aligned}$$

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

Defaults of a structural mapping

Figure 3.3 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 3.4.

Nesting

The mapping illustrated in Figure 3.3 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 [116, 79]. 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. It has the form $l=c[\textit{filterexpr}]$ where l is a target label, c is a two-dimensional cell range, and *filterexp* is a filter condition. Only cells in which *filterexp* evaluates to true are returned.

For example, in Figure 3.3 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 ob-

tained 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 has the form $l=c[\text{sort}(c_1, \text{sortorder}_1, \dots)]$ where l is a target label, c_i is a mono-dimensional range (column), and sortorder_i is the corresponding sorting order of c_i with value "ascending" or "descending".

For example, the list of orders in Figure 3.3 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

This pattern combines grouping with aggregate functions. Grouping has the form $l=\text{groupby}(c_1, c_2, \dots)$ where l is a target label and c_i is a grouping attribute (one-dimensional range).

Aggregation has the form $l=f(c)$ where l is a target label, f is an aggregate function, and c is a one-dimensional range (column).

In this example, the user wants to group the source spreadsheet in Figure 3.3(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 *column₁*, *column₂*, 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

This pattern allows users to join two tables according to a condition. It has the form $l = \text{join}(c_1, c_2, \text{joincondition})$ where c_1 and c_2 are two-dimensional ranges (tables), and *joincondition* is a condition to join two ranges c_1 and c_2 .

Suppose that the source spreadsheet in Figure 3.3(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 3.3(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. For instance `Orders =join(D2:F3, B2:C6)`, a full Cartesian product is computed between two tables: Each customer information in a row of table D1:F3 is associated with all product information in rows of table A1:C6.

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. Other set operators intersect and minus can be specified via functions *intersect(table1, table2,...)* and *minus(table1, table2,...)*, respectively. We have the following rules:

- $union(c_1) = c_1$

- $intersect(c_1) = c_1$

- $minus(c_1) = c_1$
- $union(c_1, \dots, c_n) = union(union(c_1, c_2), \dots, c_n)$
- $intersect(c_1, \dots, c_n) = intersect(intersect(c_1, c_2), \dots, c_n)$
- $minus(c_1, \dots, c_n) = minus(minus(c_1, c_2), \dots, c_n)$

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 3.3(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).

Branching

This kind of mapping allows users to map different sets of data to a target element depending on the outcome of a preset condition. It is specified via the form $l = if(condition, value_if_true, value_if_false)$. The target label **l** is assigned value *value_if_true* if *condition* evaluates to true. Otherwise, **l** is assigned to value *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 3.3(b) to indicate that whether the quantity of an order item is small or large. The following mapping is used together with the structural 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”. In this example, only the order item with quantity 150 in the second row in Figure 3.3(a) is assigned “large”.

Composition and refinement of mappings

A *complex transformation* is typically composed of multiple basic patterns of structural and value mappings. For example, the user can perform sorting and then filtering at structural level in Figure 3.3: **Orders** = A2:E50[sort(D2:D50, ascending), D2:D50=“Towel”]

Additionally, a formula specified on a label may collide with a formula inherited from one of its ancestor labels. Regarding the mapping **Orders**=A2:E50 in Figure 3.3, 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.

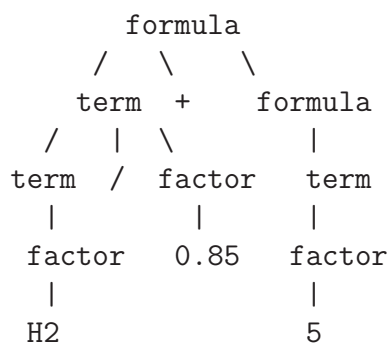
In such situation, the user can specify mapping formulas at high level (i.e. structural labels) and refine them at lower level (e.g., leaf level). These will have precedence over the values derived by the automated mechanism. For instance, mapping **Quantity**=E2:E50*10 is used to correct the problem. An interactive refinement can be performed until the user is satisfied with the example values displayed as a mapping preview.

3.3.4 User-defined functions

A user-defined function (UDF) is necessary when users want to reuse code. It avoids repetition of code which spreadsheets lack of. In this section, we describe how users can define and invoke UDFs right in spreadsheet environment rather than using an external environment and a low-level programming language such as Visual Basic. We extend the work of Johnes et al [95].

Professional programmers start defining a new function in the following steps. First, they realize that they want to define a function. Second, they define the function via a programming language. Finally, they invoke the function to test its functionality. With such steps, it requires *premature commitment* [82] in the sense that the user needs to think about the function before she writes the code [95]. However, this is unfamiliar with many spreadsheet users. Therefore, the sequence is as follows. First, the user has written some code (a formula) that works well for her task. Second, the user decides the code needs to be reused in future tasks. Third, the user reuses the code by converting the formula into a UDF via a right-click on the formula textbox. Fourth, the system guesses the number of parameters from the formula and shows up a dialogue enabling the user to configure the function (e.g., name of the function and parameters). The guess is done by parsing the formula and checking leaves of the parse tree. A leaf is a parameter if it is a cell coordinate. Finally, the original formula is replaced by the newly defined function. For consistency with built-in functions, a UDF appears in the list of functions shown up to users, once defined.

Let us consider an example of defining a UDF. After specifying the value mapping to convert format of source element **Price** to the format of target element **Price**: **Price** = D3/0.85 + 5, the user wants to turn it into a UDF. The parse tree of the formula is as follows (See Section 3.3.1 for a detailed description on the grammar of the language's formulas):



As shown in the parse tree, there is one parameter D3. The UDF is named **ConvertPrice**. The formula **Price** = D3/0.85 + 5 turns into **Price** =**ConvertPrice**(D3) and it is immediately reflected in the formula textbox.

If there is a problem with a UDF, the user may want to jump inside the function to investigate the problem. Traditional programming languages typically provide debugging techniques with the notions of breakpoints, stepping into, and stepping over. However, these concepts are unfamiliar with end-users. In order to address this issue, our system employs the technique in [95], based on *progressive evaluation* [82]. Intermediate values for each arithmetic operation (i.e., +, -, *, /) in the process of computing the function are readily shown up to the user and may be associated with any comments related to those values. These values can also be calculated based on the parse tree of the formula. The user clicks on the link associated with a target value generated by the UDF in the instance sheet to open up a new worksheet and see all intermediate values in that sheet. For example, the user wants to understand why function `Price =ConvertPrice(D3)` produces target value 9.12 from the source value 3.5 in the instance sheet. The user clicks on the link under the value 9.12 and another sheet is opened up which is an instance of function `ConvertPrice` shown as follows:

| | A | B | C | D |
|---|---|-----------|-----------------|---|
| 1 | | | | |
| 2 | | Input | 3.5 = Sheet1.D3 | |
| 3 | | AUD | 4.12 = C2/0.85 | |
| 4 | | After tax | 9.12 = C3 + 5 | |
| 5 | | | | |
| 6 | | ... | | |

The input parameter locates in cell C2 and cell C3 contains an intermediate value while the final result is in cell C4. Formula `3.5 =Sheet1.D3` indicates that the input is obtained from cell D3 of instance sheet named `Sheet1`. Cells B2–B4 contain commentary written by the user.

3.4 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 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. In some cases, it is possible to work around structural mismatches by specifying formula at a lower level, i.e., by defining mappings on atomic labels rather than on their parents. But this approach is limited and also makes the task of defining mappings more tedious since more formulas are needed.

Our solution to this problem is to allow users to organize a view of the target schema to their convenience. This is achieved by a set of *rearrangement* operations achieved in an interactive way through essentially mouse-based manipulation of the schema labels. 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 formulas should be interpreted.

3.4.1 Isomorphic view rearrangement

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

Label reordering

Label reordering consists of modifying the order of labels within a *Rcd* construct 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 3.2(b), the mapping **OrderDetails=A3:C5** would not have been possible. Instead, we would have had to specify lower level mappings: **Price =A3:A5**, **Quantity =B3:B5**, and **ProdName =C3:C5**. Both ways

produce the same interpretation. However, the benefit of label ordering becomes significant when mapping larger tables. The operation is performed by dragging a label and dropping it above or below other labels. Note that since *Rcd* labels do not introduce new nesting, children label of a *Rcd* label can be moved to the level of the *Rcd* construct itself.

Ignoring and adding labels

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 3.3 are merged with delimiter whitespace into column B containing names of customers (e.g., “Ford Prefect”); columns A, C, and D contain order identifiers, product names and quantities, respectively. 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. Labels added this way are shown in a different color and with a ‘+’ icon. No mapping formula evaluation is shown for these labels. Suppose that labels `FirstName` and `LastName` of the target schema in Figure 3.3 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`. Values of columns A2:A50, D2:D50, and E2:E50 are correctly copied into values of labels `Id`, `ProdName`, and `Quantity`, respectively. The mapping associated with `Name` must then be refined (it is incorrectly mapped with B2:B50): `Name = concatenate(B2:B50, " ", C2:C50)`.

3.4.2 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 3.4(a) is not isomorphic to the target `Quotation Request` schema (in Figure 3.3(b)), although it uses exactly the same data. The difference is that spreadsheet in Figure 3.4(a) groups data per product while the target schema groups them per order. Similar to label reordering (Section 3.4.1), TranSheet allows users to rearrange the schema in a way that matches this spreadsheet via drag&drop manipulation of labels. For example, Figure 3.4(b) shows such an rearranged view of the `Quotation Request` schema to resolve the above mismatch. Then, within a nesting level, label reordering, adding, and ignoring can be also performed as described in Section 3.4.1. Note that if a *SetOf* label is emptied from its entire children atomic labels as a result of such operations, it is displayed in a gray style and there is no mapping formulas interpreted or specified on it.

In the background, TranSheet produces tgds [79, 118] 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 3.4(b) and the original one in Figure 3.3(b):

$$\forall d \in \text{QuoteRequest.OrderDetails}, o \in d.\text{Orders} \rightarrow \exists o' \in \text{QuoteRequest.Orders} \\ | o'.\text{Order.Id} = o.\text{Order.Id}, o'.\text{Order.ShipTo.FirstName} =$$


```

o.Order.ShipTo.FirstName, o'.Order.ShipTo.LastName =
o.Order.ShipTo.LastName, [vd2 ∈ OrderDetails, o ∈ d2.Orders → ∃d' ∈
o'.Order.OrderDetails | d'.OrderLine.ProdName = d2.ProdName,
d'.OrderLine.Quantity = o.Order.Quantity]

```

3.4.3 Specializing the multiplicity of repeating labels

Mapping between spreadsheet content and an atomic label of the target schema that does not repeat is straightforward. It corresponds to a simple assignment of a unique atomic value. Mapping of repeating elements, however, is more tedious. It is easily done if the source data is suitably organized (e.g., as in the tabular form, Figure 3.3(a)) but it is less obvious in the case of the spreadsheet presented in Figure 3.2(a).

By specializing the target schema, i.e., by explicitly specifying schema to a smaller subset of possible instances, it is possible to simplify greatly the mapping of small datasets. Suppose now that the spreadsheet in Figure 3.2(a) contains three orders. Although data are simple and of small size, they cannot be mapped easily to the target schema presented in Figure 3.2(b). This target schema view allows mapping of one order, as shown in Figure 3.2(b), but it has no room for a second order. That is, users have the choice to express mapping either for one or for all orders (3 orders) described in Section 3.5, but they can not specify a mapping of exactly two orders.

We can alleviate that problem by allowing a specialization of the schema view for labels of type *SetOf*. The specialization, proposed through context menu on *SetOf* labels, lets users choose exactly how many instances of the repeating label they wish to map. Each order can then be mapped easily using distinct copies of the repeating label. For example, the spreadsheet in Figure 3.2(a) can be exported using the following schema view (we show only the relevant part) and its corresponding mapping:

```

Orders (specialized: 2 items)
  Order #1
    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, ...}
  Order #2
    Id =A7 {0525}
    ShipTo
      FirstName =left(A8,search(' ',A8)) {Arthur}
    ...

```

3.4.4 Specializing choice constructs

In the target data model defined in Section 3.2.2 appears the *Choice* construct. However, so far we do not provide interpretation of mapping formulas associated to *Choice* labels. This is because, in current TranSheet proposal, our decision is to handle choice construct only through UI manipulations of the schema view. We enable users to specialize the schema by choosing one of the options of *Choice* constructs. By doing so, choice labels are never used for specifying mapping, only one of their children label, of type τ_a , *SetOf* or *Rcd*, is used. From UI point of view, the choice is represented as a drop down list, allowing users to select one of the elements of the list, and updating the sub-tree of this label accordingly.

An alternative would be to allow users to specify a case condition associated with each option of a *Choice* as well as a mapping formula for each option. Our decision not to do so was motivated by an emphasis on simplicity over expressiveness and our empirical observation that conditional mapping toward choice construct is rather rare and altogether not a typical end-user need.

| | A | B | C | D | E |
|----|-------------------|---------------|-------------------|-----|---|
| 1 | Towel | | | | |
| 2 | 0042 | <i>Ford</i> | <i>Prefect</i> | 2 | |
| 3 | 0525 | <i>Arthur</i> | <i>Dent</i> | 1 | |
| 4 | | | | | |
| 5 | Beer | | | | |
| 6 | 0042 | <i>Ford</i> | <i>Prefect</i> | 150 | |
| 7 | 0007 | <i>Zaphod</i> | <i>Beeblebrox</i> | 300 | |
| 8 | | | | | |
| 9 | Babel Fish | | | | |
| 10 | 0525 | <i>Arthur</i> | <i>Dent</i> | 1 | |
| 11 | | ... | | | |

(a) Source spreadsheet with data grouped per product name

```

QuoteRequest
OrderDetails [ ] { 8 items }
ProdName = A1:A<next=bottom(Orders)+2> { Towel, Beer, ... }
Orders [ ] = A<bottom(ProdName)+1>:D<value=empty> (1 item)
Order
  Id { { 0042, 0525, ... }, ... }
  ShipTo
    FirstName { { Ford, Arthur, ... }, ... }
    LastName { { Prefect, Dent, ... }, ... }
    Quantity { { 2, 1, ... }, ... }

```

(b) Schema view and mapping specification

Figure 3.4: A generalized mapping specification for exporting datasets of varying sizes

3.5 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. Some built-in spreadsheet functions can be used together for generalizing mapping formulas (e.g., *offset* and *counta*) (Section 3.5.1). There are situations, however, in which such built-in functions are insufficient for expressing required mappings. We present these situations via an example and propose some novel extensions based on frequently formatting features of spreadsheets (Section 3.5.2).

3.5.1 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 3.3(a) from 150 to, say, 200 has no impact on the transformation mapping formula of Figure 3.3(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 3.3(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 3.3(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 mapping formulas. However, there are situations where built-in spreadsheet functions are not sufficient for expressing required mappings. We discuss these situations via an example in the next section.

3.5.2 New notations for generalizing mappings

The organization of data in the spreadsheet shown in Figure 3.4(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.

In the following, we propose extensions to the formula language of existing spreadsheet environments in order to address the above requirements. Our extensions are based on exploiting common formatting features used by spreadsheet templates (described in Section 3.2.1). For example, Figure 3.4(b) illustrates how these extensions can be used to export the spreadsheet in Figure 3.4(a):

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

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 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 3.3.3).

Given a label $l = (x_1, y_1) : (x_2, y_2)$, its value(s) can be obtained through the four following functions: **top**(l)= y_1 , **left**(l)= x_1 , **bottom**(l)= y_2 , and **right**(l)= x_2 .

For example, considering the mapping $l = A2 : C5$, we have $\mathbf{top}(l)=2$, $\mathbf{left}(l)=1$, $\mathbf{right}(l)=3$ and $\mathbf{bottom}(l)=5$.

Dynamic Range Length

We showed in Section 3.5.1 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 boundary coordinates. Users can refer to the location of other labels for indicating the boundaries of ranges. For instance, the range $A\langle\mathbf{bottom}(\mathbf{Order})-1\rangle:\langle\mathbf{right}(\mathbf{Order}), \mathbf{bottom}(\mathbf{Order})+5\rangle$. 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 usually rely on visual styles, such as cell font, cell color, an empty row or a border (e.g., a line surrounding a group of cell) to isolate data from each other. For example, an empty row is used in Figure 3.4(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. Such ranges have the form $\langle x_1, y_1 \rangle : \langle pred_c, y_2 \rangle$ or $\langle x_1, y_1 \rangle : \langle x_2, pred_r \rangle$ where $pred_c$ and $pred_r$ are two predicates specified over a column and a row respectively; these two predicates have the same semantic as in the *repeat...until* construct, except that there is no processing done for the column/row for which the boolean expression $pred_c$ or $pred_r$ is true. A predicate is expressed either on the value of a cell or on its style properties using the keywords **value** and **style**, respectively.

For example, to specify that a range starting at coordinate at A1 ends at an empty row, one may use $A1:A\langle\mathbf{value}=\mathbf{empty}\rangle$.

In Figure 3.4(b), both above extensions are used in the mapping formula $\mathbf{Orders}=A\langle\mathbf{bottom}(\mathbf{ProdName})+1\rangle:D\langle\mathbf{value}=\mathbf{empty}\rangle$. 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”;

Note that predicates $pred_c$ and $pred_r$ can be also used to locate a cell at varying coordinates: $\langle x, pred_r \rangle$ and $\langle pred_c, y_2 \rangle$. This is a special case of conditional range boundaries, in which a range has only one cell. For example, to find a cell containing label “FirstName” locating in column B, but with varying row, one might use: $B\langle value = \text{“FirstName”} \rangle$.

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 3.4(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 3.2.1). Using this notation, product names in Figure 3.4(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 3.4(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 3.4(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 font, color, and border (e.g., `style=bold`). For example, we could use the mapping formula `ProdName=A1:A⟨next=bottom(Orders)+2 until:style=bold⟩` to specify that the iteration stops when a cell with the bold font is found.

The user can combine the constructs presented above to transform a more complex template (e.g., the template depicted in Figure 3.4).

3.6 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 (Section 3.6.1). We then present the novelty in query generation from tgds (Section 3.6.2).

3.6.1 TGD Generation

In this section, we formally present the semantics of TranSheet. As presented so far, TranSheet mainly maps one table (e.g., sorting, filtering) or multiple tables

(e.g., join, union) to the target schema (except mappings in Figures 3.2 and 3.4). As mentioned in Section 3.3.3, to employ a structural mapping, TranSheet makes two assumptions which can be overridden. One of them is the ordering of source columns is identical to the ordering of atomic labels of the target schema. That is when traversing the spreadsheet and the target schema from left to right, the first source column corresponds to the first atomic label, the second source column corresponds to the second atomic label, and so on. Consider the simple mapping in Figure 3.3, source columns A2:A50, B2:B50, C2:C50, D2:D50, and E2:E50 correspond to atomic labels `Id`, `FirstName`, `LastName`, `ProdName`, and `Quantity` of the target schema, respectively. Let us represent these columns by the relation `Orders(Id, FirstName, LastName, ProdName, Quantity)`. Note that 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 value correspondences, using Clio’s mapping generation algorithm [79, 116], the following *tgd* (adopting the syntax used by Clip [118] to represent mappings) can be emitted for describing the structural mapping `Orders = A2:E50` between the source spreadsheet and the target schema:

```


$$\forall o \in \text{Source.Orders} \rightarrow \exists o' \in \text{QuoteRequest.Orders}, d' \in$$


$$o'.\text{OrderDetails} \mid o'.\text{Order.Id} = o.\text{Id}, o'.\text{Order.ShipTo.FirstName} =$$


$$o.\text{FirstName}, o'.\text{Order.ShipTo.LastName} = o.\text{LastName},$$


$$d'.\text{OrderLine.ProdName} = o.\text{ProdName}, d'.\text{OrderLine.Quantity} =$$


$$o.\text{Quantity}$$


```

Similarly, based on the mapping generation algorithms of Clio [79, 116] and Clip [118], the semantics of all other examples in Sections 3.3 and 3.5 can be formally described using *tgds*. In a nutshell, Clio identifies the *logical relations* (i.e., maximal tableaux [32]) of the source and target schemas. To identify them, *primary paths* are first discovered. A primary path is basically a linear tableau which is generated by enumerating all paths from the root to any intermediate node of set type in a schema. Logical relations are then generated by chasing nested referential integrity constraints against primary paths. Next, correspondences between two schemas are

used to produce mappings. Each source logical relation is paired with a target logical relation. A pair makes a mapping if it contains at least one value correspondence. To remove redundancy, if a mapping is *implied* or *subsumed* by others, then it is discarded [79]. Each mapping is described via a tgds expression. Clip [118] extends Clio’s algorithm to deal with structural correspondences explicitly, in addition to value correspondences (For a detailed description, see Section 2.5.)

We use second-order tgds of Clio [79, 116] and Clip [118] to describe the semantics of the language. In comparison with Clio [79, 116] and Clip [118], we extend tgd expressions with a collection of new functions, in addition to Skolem functions, ranging from data cleaning/data reorganizing to adding full Boolean expressivity allowing conditional branching and set difference/intersection/union in the queries (described below). This allows the languages to express popular transformation patterns that are relevant to spreadsheet-based data transformation. In the following, we present tgd expressions used to describe the semantics of mapping examples in Sections 3.3.2 and 3.3.3. We also present new functions that we introduce to tgd expressions.

Filtering. The filtering example presented in Section 3.3.3 can be described by the following tgd:

```

 $\forall o \in \text{Source.Orders} \mid (o.\text{ProdName} = \text{'Towel'}) \ \&\& \ (o.\text{Quantity} > 1) \rightarrow$ 
 $\exists o' \in \text{QuoteRequest.Orders}, d' \in o'.\text{OrderDetails} \mid o'.\text{Order.Id} =$ 
 $o.\text{Id}, o'.\text{Order.ShipTo.FirstName} = o.\text{FirstName},$ 
 $o'.\text{Order.ShipTo.LastName} = o.\text{LastName}, d'.\text{OrderLine.ProdName} =$ 
 $o.\text{ProdName}, d'.\text{OrderLine.Quantity} = o.\text{Quantity}$ 

```

Grouping with aggregation. The tgd for the grouping with aggregation example presented in Section 3.3.3 is as follows:

```

 $\exists \text{groupby}, \text{count}, \text{max}(\forall o \in \text{Source.Orders} \rightarrow \exists o' \in \text{Target.Orders} \mid o'$ 
 $= \text{groupby}(\perp, o.\text{OrderId}, o.\text{FirstName}, o.\text{LastName}) \ o'.\text{Order.Id} = o.\text{Id},$ 
 $o'.\text{Order.FirstName} = o.\text{FirstName}, o'.\text{Order.LastName} = o.\text{LastName},$ 
 $o'.\text{Order.ProdName} = \text{count}(o.\text{ProdName}), o'.\text{Order.Quantity} =$ 
 $\text{max}(o.\text{Quantity}))$ 

```

Join. The tgd corresponding to the join example presented in Section 3.3.3 is:

$$\begin{aligned} &\forall o \in \text{Source.Orders}, c \in \text{Source.Customers} \mid o.\text{Id} = c.\text{Id} \rightarrow \exists o' \in \\ &\text{QuoteRequest.Orders}, d' \in o'.\text{OrderDetails} \mid o'.\text{Order.Id} = c.\text{Id}, \\ &o'.\text{Order.ShipTo.FirstName} = c'.\text{FirstName}, o'.\text{Order.ShipTo.LastName} = \\ &c'.\text{LastName}, d'.\text{OrderLine.ProdName} = o.\text{ProdName}, \\ &d'.\text{OrderLine.Quantity} = o.\text{Quantity} \end{aligned}$$

Refinement of mappings. The tgd corresponding to the mapping refinement example presented in Section 3.3.3 is as follows:

$$\begin{aligned} &\forall o \in \text{Source.Orders} \rightarrow \exists o' \in \text{QuoteRequest.Orders}, d' \in \\ &o'.\text{OrderDetails} \mid o'.\text{Order.Id} = o.\text{Id}, o'.\text{Order.ShipTo.FirstName} = \\ &o.\text{FirstName}, o'.\text{Order.ShipTo.LastName} = o.\text{LastName}, \\ &d'.\text{OrderLine.ProdName} = o.\text{ProdName}, d'.\text{OrderLine.Quantity} = \\ &o.\text{Quantity} * 10 \end{aligned}$$

Examples in Figures 3.4 and 3.2. In the mapping depicted in Figure 3.4, 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 3.4(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 3.4(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 [104]. It is different from the spreadsheet in Figure 3.3(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 Fig-

ure 3.4 (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 | <i>ProdName</i> | <i>Id</i> | <i>FirstName</i> | <i>LastName</i> | <i>Quantity</i> | |
| 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:

```

 $\forall o \in \text{Source.Orders} \rightarrow \exists o' \in \text{QuoteRequest.OrderDetails}, d' \in$ 
 $o'.\text{Orders} \mid o'.\text{ProdName} = o.\text{ProdName}, d'.\text{Order.Id} = o.\text{Id},$ 
 $d'.\text{Order.ShipTo.FirstName} = o.\text{FirstName}, d'.\text{Order.ShipTo.LastName} =$ 
 $o.\text{LastName}, d'.\text{Order.Quantity} = o.\text{Quantity}$ 

```

With respect to the mapping in Figure 3.2, the source spreadsheet (Figure 3.2(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`.

New functions in `tgd` expressions

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 tgdc for the mapping example in Section 3.3.3 is:

```

 $\exists \text{sort}(\forall o \in \text{Source.Orders} \rightarrow \exists o' \in \text{QuoteRequest.Orders}, d' \in$ 
 $o'.\text{OrderDetails} \mid o' = \text{sort}(\perp, o.\text{ProdName}, \text{ASC}, o.\text{Quantity}, \text{DESC}),$ 
 $o'.\text{Order.Id} = o.\text{Id}, o'.\text{Order.ShipTo.FirstName} =$ 
 $o.\text{FirstName}, o'.\text{Order.ShipTo.LastName} = o.\text{LastName},$ 
 $d'.\text{OrderLine.ProdName} = o.\text{ProdName}, d'.\text{OrderLine.Quantity} =$ 
 $o.\text{Quantity})$ 

```

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 tgdc corresponding to the mapping example in Section 3.3.3 is:

```

 $\exists \text{if}(\forall o \in \text{Source.Orders} \rightarrow \exists o' \in \text{QuoteRequest.Orders}, d' \in$ 
 $o'.\text{OrderDetails} \mid o'.\text{Order.Id} = o.\text{Id}, o'.\text{Order.ShipTo.FirstName} =$ 
 $o.\text{FirstName}, o'.\text{Order.ShipTo.LastName} = o.\text{LastName},$ 
 $d'.\text{OrderLine.ProdName} = o.\text{ProdName}, d'.\text{OrderLine.Quantity} =$ 
 $o.\text{Quantity}, d'.\text{OrderLine.Description} = \text{if}(o.\text{Quantity} > 20, \text{'large'},$ 
 $\text{'small'}))$ 

```

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 tgdc for the mapping example in Section 3.3.3 is:

```

 $\exists \text{union}(\forall o1 \in \text{Source.Orders1}, o2 \in \text{Source.Orders2} \rightarrow \exists o' \in$ 
 $\text{QuoteRequest.Orders} \mid o' = \text{union}(o1, o2))$ 

```

To support various patterns of value mapping, we introduce a collection of Excel-like functions in tgdc expressions, such as *left*, *right*, *search*, *len*, and *concatenate*. For example, the tgdc for the mapping example in Section 3.4.1 is:

```

 $\exists \text{left}, \text{search}, \text{right}, \text{len} (\forall o \in \text{Source.Orders} \rightarrow \exists o' \in$ 
 $\text{QuoteRequest.Orders}, d' \in o'.\text{OrderDetails} \mid o'.\text{Order.Id} = o.\text{Id},$ 
 $o'.\text{Order.ShipTo.FirstName} = \text{left}(o.\text{Name}, \text{search}(' ', o.\text{Name})),$ 
 $o'.\text{Order.ShipTo.LastName} = \text{right}(o.\text{Name}, \text{len}(o.\text{Name}) - \text{search}('$ 
 $', o.\text{Name})), d'.\text{OrderLine.ProdName} = o.\text{ProdName}, \dots)$ 

```

3.6.2 Query Generation

Once the tgds have been generated, they are used to produce executable query XQuery for transformation as presented in [79, 118] (See Section 2.5). We first summarize data translation algorithms of Clio and Clip and important properties of these algorithms. We then describe the new extensions that we make to these algorithms.

Basic steps and properties of query generation

The main idea is that each tgd is translated in to a query (i.e., XQuery script). After that, resulting queries are unioned to obtain final result [116]. Let d be a tgd from source logical relation A to target logical relation B , the individual query generated for d is divided into two steps. First, it materializes the flat relational view A and projects on the attributes that are used by d . After some renaming from attributes of A to attributes of B , the result is basically a projection of B . Second, it nests the result according to the structure of the target and creates new values for undetermined attributes using Skolemization.

The authors in [79] extend the above algorithm to deal with nested mappings. This algorithm takes as input a nested mapping M and produces an XQuery FLWOR expression as output. Each sub-mapping of M is translated into one nested FLWOR expression of F . F has the following structure: a “for” clause captures the iteration implied by every universally quantified variable of M ; a “where” clause captures the join and filtering predicates; a “return” clause constructs the XML items for the target schema elements mentioned in the existentially quantified part of the mapping; elements bound to some of the variables defined in the for clauses are

copied to the proper positions according to the values mappings expressed in the mapping M . In turn, the sub-mappings of M recursively replicate this structure. Clip [118] presents some extensions to support minimum-cardinality assumption, grouping, and aggregate functions.

The algorithm has the following crucial properties [116]:

- The resulting target instance is in Partition Normal Form (PNF) [33]. This means that in any set, at any nesting level of the target instance, the atomic type attributes functionality determine the set type attributes. PNF is important because it ensures that the redundancy that can occur due to nesting has been minimized.
- More importantly, the generated query populates the target with the source data that is intended to be mapped by the logical mapping (i.e., information preservation). In other words, given a source instance, the algorithm generates a single *canonical universal solution* that is the result of the logical mapping [71] (See Section 2.6 for a detailed description on universal solution). Indeed, the algorithm described above is a special case of the chase procedures of the general data exchange problem presented in [71].

Since our query generation algorithm is developed on top of the aforementioned algorithm, it guarantees that each target document of TranSheet corresponds to a canonical universal solution.

Novelty in query generation

In what follows, we 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 *tgds* 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 3.3.3 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 tgdl 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 3.3.3 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 tgdl

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 3.3.3 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); }
```

3.7 Implementation

Architecture. TranSheet has been implemented as an Excel plug-in using C# 3.0 and Visual Studio 2008. Figure 3.5 depicts the architecture of TranSheet with the following main components: (i) GUI enables users to specify mappings 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 3.6.1); (iii) Query generation engine generates XQuery from input tgds (Section 3.6.2); (iv) Execution engine is responsible for executing input XQuery and then returning the transformation result to GUI for validation. TranSheet currently employs the open-source execution engine Saxon ¹.

User interface. The user interface of TranSheet is shown in Figure 3.6. 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. For a detailed demonstration on how TranSheet works, see Appendix A.2.

Similar to [123], for simple mappings like copying (e.g., mappings `Id =A1`, `ProdName =C3:C5`, `OrderDetails =B3:C5` in Figure 3.2), 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 complex transformation operators, such as filtering, sorting, merging, splitting, and join, TranSheet provides a collection of *form-based wizards* that help users specify mapping graphically, rather than writing complicated formulas from scratch. Details with examples about these wizards can be found in Chapter 5.

Warnings. There are two level of warning display in the user interface: warnings at the label level and warning at document level:

Warnings at label level

Some schema constraints are attached to particular schema labels. For example, labels expect a particular type, or some label of a tuple should have the same cardinality as the other label of the tuple (See Section 3.3.2).

For instance, if a label expects an integer and its mapping formula is set to ‘=A1’,

¹<http://saxon.sourceforge.net/>

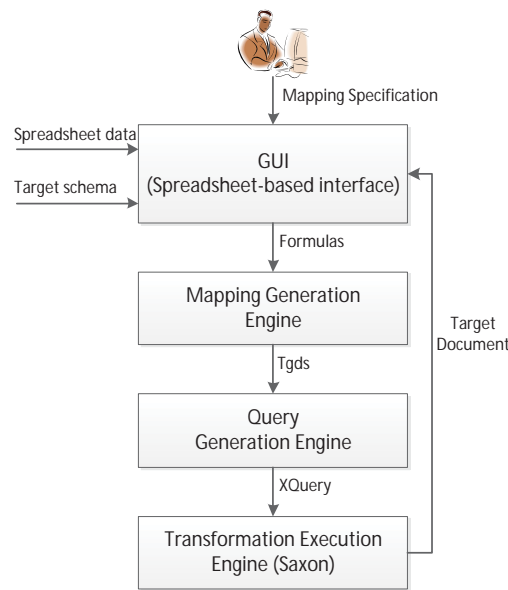


Figure 3.5: Architecture of TranSheet for the spreadsheet-based data transformation language

a warning message “#type mismatch: integer value expected” is displayed as long as A1 does not contain the correct data type. When there is warnings generated during formula evaluation and validation, the result of the formula evaluation is displayed (i.e., in this case, the content of cell A1).

Warnings at document level

TranSheet dynamically generate XML document for each modification of either the spreadsheet or the mapping specification. This gives an immediate feedback on the final form of the message. Although displaying XML documents may be of little interest to end-users, we use this document for a final validation against the schema. It allows highlighting constraints violation that does not appear in our simplified schema representation.

For example, XML Schema allows defining identifier attributes as well as references to these identifiers. To check for such constraints, we use a generic schema validation tool and display its output in a separate zone of the UI. Our plan is

to progressively take into account as much constraints as possible directly at the schema or element level, as well as for the interpretation of mapping formula, so that these errors get displayed in their closest related location.

3.8 Experiments

So far, within the scope of this chapter, we mainly present the semantics and foundation of the language, rather than its usability features. Thus, in this section, we only 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 (Excel XML Mapping) [123] and IBM ManyEyes [136, 8] in the visualization context, which maps spreadsheets to visualization types (Section 3.8.1). 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 (Section 3.8.2).

3.8.1 Expressiveness

Experimental setup. We selected 5 real data sets from the public repositories of Australian government agencies ² (4 large data sets) and ManyEyes [8] (1 data set); each data set has column headers. The *NSW Crime* dataset is a table showing crimes level in New South Wales - Australia by offence type, month and local government area from 1995 to 2009 with 184 columns and 10541 rows. The *Economic Stimulus Package* data set contains national building - economic stimulus plan projects of Australia with 29 columns and 4271 rows. The *School Locations* data set consists of addresses of schools in Victoria - Australia with 12 columns and 2294 rows. The *Frog Atlas* contains various kinds of frogs in South Australia recorded at different places with 13 columns and 6732 rows. The *expensive cities* data set indicates the most expensive cities in the world from 2002 to 2009 with 9 columns and 144 rows.

²<http://data.australia.gov.au/>

| Mapping Scenario | Data Set | Visualization Type | Description |
|---------------------------|---------------------------|--------------------|---|
| Copying | NSW Crime | Pie Chart | Visualize the top 10 number of crimes in December 2009 using a pie chart. |
| Merging | School Locations | Map | Merge street, town, and post code information to put school locations on a map. |
| Derivation | Frog Atlas | Time Line | Visualize the names and descriptions of frogs along with their sighting dates using a time line, in which format dd/mm/yyyy (Australia format) is converted to format mm/dd/yyyy (US format). |
| Splitting | Expensive Cities | Map | Put the 143 most expensive cities in 2009 on a world map. Split the city information in the data set to get country names for visualization. |
| Sorting | Economic Stimulus Package | Pie Chart | Visualize the projects along with corresponding government funding for each project in the descending order of funding using a pie chart. |
| Filtering | NSW Crime | Scatter Plot | Use a scatter plot find correlation between crimes in November 2009 and crimes in December 2009, but only focus on area Marrickville. |
| Grouping with aggregation | Economic Stimulus Package | Pie Chart | Group the data set by program type, and then average funding for each program type; visualize program types with their corresponding average fundings using a pie chart. |
| Nesting | NSW Crime | Bar Chart | Visualize a bar chart of different types of crimes in area Botany Bay in October 2009. |

Table 3.2: Mapping Scenarios

Our test schemas are five structurally different visualization types *Pie Chart*, *Scatter Plot*, *Bar Chart*, *Map* of ManyEyes [136], and *Time Line* of SIMILE project ³. Schemas corresponding to some visualization types are shown in Figure 3.1. All data sets and schemas can be found on our web page ⁴.

Eight popular mapping scenarios commonly used by the ManyEyes community for visualization [8, 136] are considered in this experiment and are summarized in Table 3.2. 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 Excel XML Mapping and ManyEyes. For each mapping scenario in Table 3.2, 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 Excel XML Mapping and ManyEyes need multiple manipulations on data sets to accomplish mapping scenarios (summarized in Table 3.3). Excel XML Mapping generally requires fewer manipulations than ManyEyes and multiple manipulations of Excel XML Mapping can be done using graphical wizards. This is because Excel provides many advanced features to support data manipulation. However, Excel XML Mapping cannot implement mapping scenario nesting. While Excel XML Mapping supports only one nesting level in the target schema, a bar chart consists of two nesting levels.

Excel XML Mapping supports copying by dragging target attributes onto columns containing local government area and crimes in December 2009. In the case of ManyEyes, while the column containing local government area is selected from 4 candidates to map with the text target attribute, the column containing crimes in December 2009 is selected from 180 candidates to map with the numeric target at-

³<http://www.simile-widgets.org/>

⁴<http://cgi.cse.unsw.edu.au/~vthung/>

| Mapping Scenario/Tool | EXM | ManyEyes |
|---------------------------|---------------------|------------------------|
| | Source Manipulation | Source Manipulation |
| Copying | 10530RD | 10530RD |
| Merging | 1CI+2293VC | 1CI+2293VC |
| Derivation | 6731VC | 6731VC |
| Splitting | 123RD+1CI+20VC | 123RD+1CI+20VC |
| Sorting | 1DS | 4270RD+4270RI |
| Filtering | 1DF | 10473RD |
| Grouping with aggregation | 2CI+5RI+10VC | 2CI+5RI+10VC+29CD |
| Nesting | Not supported | 2CI+69RI+138VC + 184CD |

Table 3.3: Source manipulation operations of EXM and ManyEyes in implementing mapping scenarios

tribute. However, 10530 unwanted rows must be deleted for both tools. Although Excel offers range notation, Excel XML Mapping selects by default the entire column, even if a specific range of the column is selected. TranSheet supports copying via either range formulas or drag-and-drop operations.

To implement derivation, Excel XML Mapping and ManyEyes require 6731 values in the sighting date column must be converted to the mm/dd/yyyy format. Similarly, a new column is inserted and its values are changed in the case of the splitting or merging scenario to store splitting/merging values. TranSheet supports derivation, merging, and splitting via applying functions on range formulas. For instance, function *concatenate* is used to merge three columns containing street, town, and postcode into a single address and then this address is mapped to the text target attribute of the visualization type map.

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 government funding in descending order, for instance, the user must manually select, cut and paste all required rows individually. To filter area Marickville, the rows containing other areas are deleted from the data set. TranSheet supports filtering and sorting by performing struc-

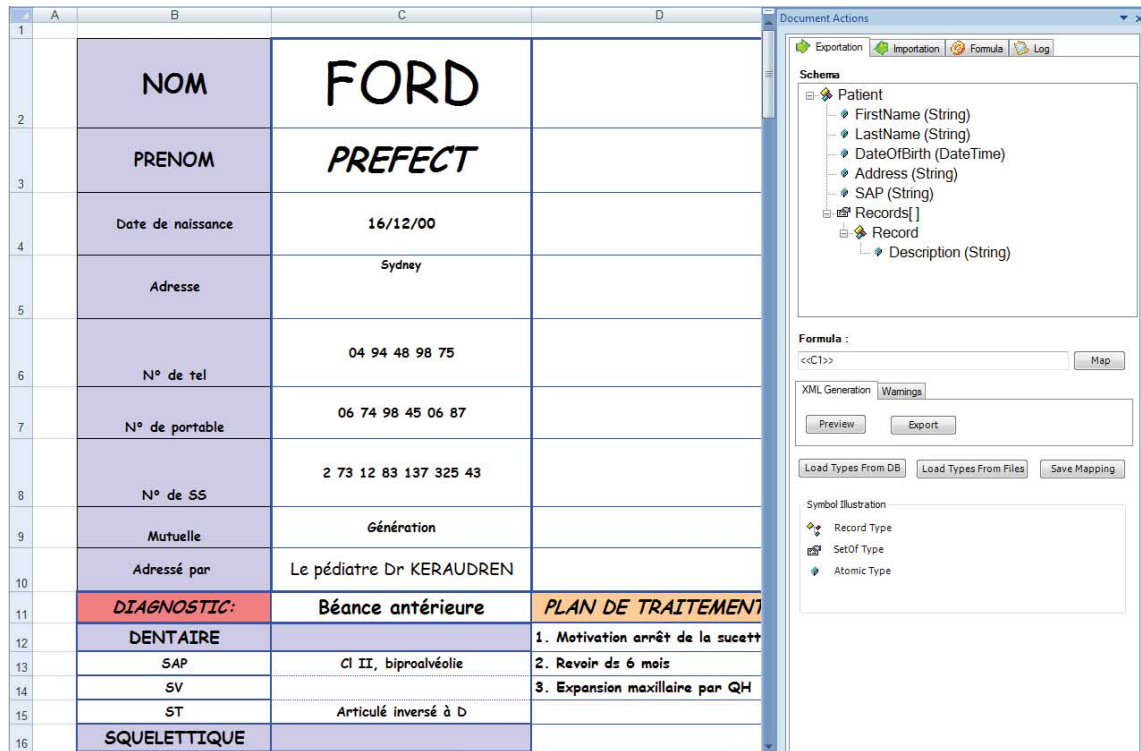
tural mappings via formulas or wizards. For instance, the structural mapping `Dots=A2:GB10541[B2:B10541="Marickville"]` is used for filtering.

To implement grouping with aggregation using Excel XML Mapping, a new table is created with 2 columns and 5 rows, in which each row contains a program type and its average funding. To perform grouping, ManyEyes selects by default three text columns containing project type, program type, and agency as grouping candidates and calculates totals for column funding. Moreover, it supports only one grouping attribute and one aggregate function to compute totals. As a result, this mapping scenario cannot be implemented using the default grouping of ManyEyes. Instead, like Excel XML Mapping, the user must create a new table and then deletes 29 old columns to avoid default grouping. TranSheet supports this mapping scenario by grouping the data set according to program type and then using the aggregate function *average*.

Since Excel XML Mapping does not support nesting, we only focus on ManyEyes. To implement this scenario using ManyEyes, a new table with is created with 2 columns and 69 rows, where each row contains different types of crimes in October 2009, except the header row. In addition, 184 old columns must be deleted. TranSheet supports nesting by mapping range formulas to 2 attributes of the bar chart. The title of the bar chart is assigned with value “Botany”.

3.8.2 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 3.6, in which the personal information



The screenshot displays the TranSheet user interface. On the left is an Excel spreadsheet with columns A, B, C, and D. The data is organized into rows with labels in column B and values in column C. The last row (row 16) contains a list of treatments in column D. On the right is a 'Document Actions' sidebar with a 'Schema' tree, a 'Formula' input field, 'XML Generation' buttons, and a 'Symbol Illustration' section.

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

Figure 3.6: TranSheet user interface

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 Excel XML Mapping because ManyEyes is limited to predefined schemas for visualization. With respect to 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 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 3.6). 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 3.6, 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 3.5.2) and the second problem through the expression of relative locations of varying coordinate cells (Sections 3.5.2 and 3.5.2). 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)

3.9 Related Work

3.9.1 Transformation Languages and Mapping Tools

A language for the description of spreadsheet content as a series of relational tables is proposed in [104, 60]. Once defined, the schema can be used with either low-level transformation languages (e.g., XSLT/XQuery) or visual mapping tools (e.g., Clio [79], Clip [118], +Spicy [111], and Altova MapForce [2]) to perform transformation. However, spreadsheet users must learn a new complex language to perform transformations 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.

3.9.2 Exportation of spreadsheet data

Several existing approaches support the exportation of spreadsheet data to structured formats [123, 53, 136, 5]. 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.

3.9.3 Schema Mapping and Data Exchange

Data exchange studies how to transform an instance of a source schema into data structured under a target schema and the restructured data is materialized. In data exchange, given a source instance, there are many possible solutions for the target [72]. 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 [79, 118], TranSheet introduces a collection of new functions in tgd expressions to cover numerous transformation patterns provided by other transformation languages and mapping tools [37]. TranSheet also extends a previous query generation algorithm [118] to generate XQuery for these new functions.

It is shown in [79] that target data may contain a lot of redundancy when using existing mapping generation algorithms [79, 116]. Fagin et al. [73] introduce the notion of core solutions which are optimal solutions for the data exchange problem. The post-processing algorithm for computing core solutions in [73] can be adopted

by our language to remove such redundancy.

Currently, a formal description of the exact class of mapping scenarios that can be handled by *tgd*-based mapping systems (including *TranSheet*) is still missing [143].

3.9.4 Spreadsheet-based data access and manipulation

The spreadsheet programming paradigm is leveraged in [106, 135] to simplify specification of SQL queries using formulas. *Mashroom* [139] 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, rather than extracting data from the spreadsheet, including macros and formulas. The recently proposed file format, namely OpenXML [125], allows external applications to extract data from Excel 2007 files (XLSX).

Our previous work [102] focused on importation external data into spreadsheets and proposed a number of widgets to present such data. Some generic notations described in Section 3.5 are used to bind data to those widgets. Instead, in this chapter, 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.

3.9.5 Domain specific and data description languages

Various high level languages have been developed to address the generic problem of accessing and representing data stored in a poorly structured way using higher level conceptual representation. In [76], the authors propose to formalize a family of Data Description Languages (DDL) in a framework called Data Description Calculus. This framework allows correctness criteria to be expressed and proved.

The DDL family of language however is geared toward parsing of data represented as a sequence of bytes (ad-hoc data). Spreadsheets are not only more struc-

tured than ad-hoc data, but they are very special in the spatial organization of data they offer. They claim for a different family of data description language.

To the best of our knowledge, the first proposal of spreadsheet-specific DDL is in [104]. The language they propose allows to specify the semantics of a spreadsheet content as well as materialize the corresponding view. This language and the one we propose in this chapter share many similar concepts (e.g., the use of variable ranges associated with concepts is equivalent to our use of labels associated with ranges).

With respect to this seminal work, our contributions are: (i) we target the XML nested data model rather than the relational model and focus on expressing a mapping toward a *given* target schema rather than describing the spreadsheet content as-is; (ii) the language proposed in [104] is specialized to pattern description, meaning it is not possible to express irregular layouts or to simplify the expression of layout when there are very few instances. TranSheet on the other hand allows freely choosing to specify an ‘instance level’ or a ‘pattern level’ mapping, depending on the actual spreadsheet layout or, more likely, on the user expertise. Moreover, schema view specialization allows to handle irregular layout cases; (iii) and this is probably the most important, we claim that TranSheet is more suited to end-users: while users in [104] have to make an explicit use of iteration variables, TranSheet abstracts this mechanism away from users thanks to the formula interpretation mechanism. Moreover, TranSheet combines visual programming (through view manipulation) with formula-based specification where formulas are very alike spreadsheet formulas.

3.10 Summary

In this chapter, 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 popular transformation patterns that are relevant to spreadsheet-based data transformation using spreadsheet formulas and functions. 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. The experimental results show that TranSheet is flexible and expressive enough to support numerous practical spreadsheet-based transformation scenarios.

We have found the proposed language expressive enough for a large range of practical scenarios. Moreover, the expressiveness of the language can easily be extended through new mapping primitives. Since spreadsheets have proved a very versatile medium for data exchange, we believe this work has a great potential for both desktop and Web-based applications.

Chapter 4

Spreadsheet-based Data Transformation Reuse

Although the reuse of previously specified mappings promises a significant reduction in manual and time-consuming transformation tasks, its potential has not been fully realized in current approaches and systems. In this chapter, we study the problem of reuse in transforming spreadsheet data to structured formats based on extending the language presented in Chapter 3. We formulate the problem and propose a solution that relies on the notions of spreadsheet templates, mapping generalization, and similarity join. Given a spreadsheet instance that is being mapped to the target schema, we recommend a list of previously specified mapping formulas that can be potentially reused for the instance. We implemented a prototype of the proposed solution and evaluated its performance via synthetic datasets.

4.1 Introduction

It has been known that data transformation is a labour-intensive and error-prone process, which includes the various steps, namely schema matching, mapping specification, code generation, and transformation execution [85, 127] (See Section 1.1.2 for more details). Thus, it is useful to reuse previously specified mappings as much as possible to save time and avoid duplication of efforts [119]. However, most exist-

ing work on data transformation mainly focuses on the development of an ad-hoc program that can handle only exactly one data source without explicit intent for future reuse [128]. As a consequence, it is cumbersome as regards to maintenance and extensibility when dealing with new data sources.

In this chapter, we consider the problem of reuse in transforming spreadsheet data to structured formats. The problem is challenging because of several reasons: (i) Spreadsheets do not impose constraints on the data layout and data is, therefore, not organized in a pre-defined way. Given two spreadsheet instances, it is hard to automatically uncover if they are similar in terms of structure; (ii) A mapping of a spreadsheet instance to the target schema is only applied exactly to this instance, not to other instances with similar structure that also need to be mapped to the target schema.

Figure 4.1 shows the transformation scenario used throughout the whole chapter for illustration purpose. We consider the three spreadsheets (in Figures 4.1(a), 4.1(b), and 4.1(c)) containing similar data that must be mapped to the target schema (Figure 4.1(d)). The instance in Figure 4.1(a) is a *table presentation* [101] with headers in the first row and data in subsequent rows. The instance in Figure 4.1(b) is a vertical *repeater presentation* [101] where different values of tuples (**Dept**, **ID**, **Name**) are presented vertically, each of which is separated by two empty rows. The instance in Figure 4.1(c) is also a vertical repeater presentation where different values of tuples (**Dept**, **ID**, **Address**, **Name**) are presented vertically, each of which is separated by one empty row. Note that spreadsheet data in Figure 4.1(b) is similar to the one in Figure 4.1(c) in terms of structure, but it does not include address information of employees.

Suppose that the two spreadsheets in Figures 4.1(a) and 4.1(b) are already mapped to the target schema. Now the user wants to map the spreadsheet in Figure 4.1(c) to this schema. The important question to ask is if previous mapping efforts can be reused for this new mapping. The structure of the spreadsheet in Figure 4.1(b) is similar to the one of the spreadsheet in Figure 4.1(c) so it is desirable that we can reuse the mapping used to transform the instance in Figure 4.1(b) to the target format for the new mapping. In what follows, we review two related

| | A | B | C | D |
|---|------------|------|---------|-----------|
| 1 | Dept | ID | Name | Address |
| 2 | IT | 1111 | Ann | Sydney |
| 3 | Marketing | 2222 | John | Melbourne |
| 4 | Sales | 3333 | Jeffrey | Adelaide |
| 5 | Accounting | 4444 | Steve | Perth |
| 6 | | | | |

(a)

| | A | B | C |
|---|------|-----------|---|
| 1 | Dept | IT | |
| 2 | ID | 1111 | |
| 3 | Name | Ann | |
| 4 | | | |
| 5 | | | |
| 6 | Dept | Marketing | |
| 7 | ID | 2222 | |
| 8 | Name | John | |
| 9 | | | |

(b)

| | A | B | C |
|----|---------|-----------|---|
| 1 | Dept | IT | |
| 2 | ID | 1111 | |
| 3 | Address | Sydney | |
| 4 | Name | Ann | |
| 5 | | | |
| 6 | Dept | Marketing | |
| 7 | ID | 2222 | |
| 8 | Address | Melbourne | |
| 9 | Name | John | |
| 10 | | | |

(c)

Target
Emp [1..*]
 Dept =B1 {/I}
 EmployeeID =B2 {1111}
 EmployeeName =B3 {Ann}
 Address?
 (d)

Figure 4.1: Motivating example: (a) Employees organized in a table presentation; (b) Employees organized in a vertical repeater presentation without address information; (c) Employees organized in a vertical repeater presentation with address information; (d) The target schema.

state-of-the-art approaches for dealing with this scenario.

The first approach, namely schema-based, helps users specify a schema of a spreadsheet [104]. Then, either low-level transformation languages (e.g., XSLT/XQuery) or visual mapping tools (e.g., Clio [79], Clip [118], +Spicy [111], Altova MapForce [2]) is used for specifying transformation at *schema level*. Finally, the spreadsheet instance is translated into an instance of the target schema. However, users must learn a new language, e.g., by creating correspondences between the source and target elements and annotating these correspondences with one or more unfamiliar functions (e.g., functions of XSLT/XQuery or .NET Framework) in the case of mapping tools [127]. This flowchart-like mapping interface is typically cluttered when schemas are large and mappings are complex [126]. In contrast, spreadsheet users are familiar with formulas and an incremental approach to building applications with instant feedback at each step [96].

The second approach, namely column-based, allows users to specify mappings between target elements and spreadsheet columns via drag-and-drop operations [124,

52]. The user can select a target atomic element and drag it onto a source column to specify a mapping. However, this approach offers no reuse support. For example, to map the spreadsheet in Figure 4.1(a), the user can select target atomic elements `Dept`, `EmployeeID`, `EmployeeName`, `Address` and drag them onto source columns A1:A5, B1:B5, C1:C5, D1:D5, respectively. To map the spreadsheet in Figure 4.1(b), the user must modify the presentation of this spreadsheet in order to conform to the structure shown in Figure 4.1(a) (i.e., table presentation) and then repeat the steps described in the case of the spreadsheet in Figure 4.1(a). Regarding the spreadsheet in Figure 4.1(c), again, the user must modify this spreadsheet as in the case of the instance in Figure 4.1(b). It is a tedious and time-consuming process with no reuse support.

In Chapter 3, we developed a spreadsheet-like formula mapping language that enables users to specify mappings between spreadsheet data and the target schema. In this chapter, we extend this language allowing users to reuse previously specified mapping formulas for a new spreadsheet instance that needs to be mapped to the target schema. To the best of our knowledge, the problem of spreadsheet-based transformation reuse has not been addressed before in the setting we consider here. More specifically, to address this problem, we make the following main contributions:

- We formulate the problem of spreadsheet-based transformation reuse as a variant of *similarity join* [43, 57, 141], which is a well-known similarity search problem that finds all pairs of objects whose similarity is above a given threshold (Section 4.2).
- We define spreadsheet templates that are used to characterize spreadsheet structures. We propose techniques to infer a template from an existing spreadsheet based on common spreadsheet presentation patterns. We then generate the string-based representation of an inferred template (Section 4.3).
- We propose an algorithm to recommend previously specified mappings for a new spreadsheet instance that needs to be mapped to the target schema. This relies on computing similarity between string-based representations of templates (Section 4.4).

- We design a repository to organize mapping information. We implemented a prototype of the proposed solution. We then evaluated the performance and effectiveness of the solution. The experimental results show the viability and usefulness of our approach (Sections 4.5 and 4.6).

The rest of this chapter is organized as follows. Section 4.2 formulate the spreadsheet-based data transformation reuse problem. Next, Section 4.3 formally defines spreadsheet templates and infers templates from existing spreadsheets. The reuse recommendation algorithm is described in Section 4.4. Section 4.5 presents the prototype implementation. Section 4.6 evaluates the performance of the proposed solution. We discuss related work in Section 4.7 and conclude in Section 4.8.

4.2 Problem Definition

Given a schema T and a spreadsheet instance I , the corresponding specified mapping M_I from I to T is stored as a tuple (M_I, I, P_I, T) in the mapping repository; P_I is a template of I ; M_I consists of template-level mappings (See Section 3.5) that are specified to transform I to a target instance conforming to T . We denote Γ_T as a collection of all tuples containing previously specified mappings from past spreadsheet instances to T . The main technical problem can be formally stated as follows.

Definition 4.2.1 *Given an instance J conforming to template P_J that is currently being mapped to schema T , find in Γ_T all tuples (M_I, I, P_I, T) such that similarity between P_I and P_J is greater than or equal θ where θ is a predefined normalized threshold ($0 < \theta \leq 1$).*

As presented later, a template is represented as a string generated from a context-free grammar. Therefore, similarity between P_I and P_J can be characterized by a string similarity function sim : $sim(P_I, P_J) \geq \theta$. It is also worth noting that the size of Γ_T ($|\Gamma_T|$) may be large (e.g., a few hundred thousand mappings) so we need a more efficient approach, rather than directly comparing P_J with all templates

in Γ_T using a similarity function (i.e., pair-wise comparison), which is costly when template lengths are large.

Given two sets of strings R and R' , *similarity join* [43, 57, 141] finds in all pairs (x, y) ($x \in R$ and $y \in R'$) such that $\text{sim}(x, y) \geq \theta$. *Self-join* is a special case of similarity join when $R = R'$. Instead, given a new string x , we find all $y \in R$ (R') such that $\text{sim}(x, y) \geq \theta$. The efficient approach to similarity join is a *filtering* (probe the inverted lists and use the filtering methods to eliminate as much as possible false candidates) and *verification* (check each candidate to find out if the threshold is satisfied) process [141], which is also applicable to the spreadsheet-based transformation reuse problem.

A spreadsheet S may contain multiple instances, not only one: $S = \{I_1, \dots, I_n\}$. For example, the user can put three instances shown in Figures 4.1(a), 4.1(b), and 4.1(c) in one spreadsheet, instead of three separate spreadsheets. We assume the existence of procedures to identify and separate I_i ($1 \leq i \leq n$) from S (See [34] for an example). Mapping from S to the target schema T can be, therefore, divided into individual mappings: $(I_1, T), \dots, (I_n, T)$. Thus, in this chapter, we assume that S contains only one instance.

4.3 Spreadsheet Template

In Section 4.3.1, we formally define the template description language. Section 4.3.2 presents template inference techniques and how to generate the string-based representation of an inferred template.

4.3.1 Template Description Language

In many cases, spreadsheets evolve in a number of predictable ways and various spreadsheets tend to emerge from a common pattern. Structure of spreadsheets can be characterized via the notion of template. We use a variation of the language VITSL developed in [69] with the following context-free grammar to describe templates:

1. $Temp ::= C \mid C^{\rightarrow} \mid Temp \mid Temp$
2. $C ::= B \mid B^{\downarrow} \mid C \mid C$
3. $B ::= F \mid B - B$
4. $F ::= \varepsilon \mid const \mid \beta \mid \Phi(F, \dots, F)$

N is a set of non-terminal symbols: $N = \{Temp, B, C, F\}$. Σ is a set of terminal symbols: $\Sigma = \varepsilon \cup const \cup \beta \cup \Phi$ and $\varepsilon \cap const \cap \beta \cap \Phi = \emptyset$. S is the start symbol: $S = Temp$.

Template $Temp$ is a table given by a horizontal composition (\mid) of fixed columns (C) or expandable groups of columns (C^{\rightarrow}). A column (C) is given by a vertical composition ($-$) of fixed cells (B) or expandable groups of cells (B^{\downarrow}). A cell (B) is given by a formula (F), which consists of an empty value (ε), a constant label ($const$), a basic type (β) (e.g., int, string) of a data cell, or a function ($\Phi(F, \dots, F)$). Note that cells of a template can be basically classified according to their content into four types: empty cell, label cell (e.g., headers), data cell, and formula cell (i.e., computation cell) [34].

All columns of a template have to vertically align (i.e., same height and same expandable groups of cells). An expandable group of columns is called a horizontal expandable group or *hex group* for short. An expandable group of cells is called a vertical expandable group or *vex group* for short. For comparison purpose, we replace the hex group symbol C^{\rightarrow} and the vex group symbol B^{\downarrow} by $[C]$ and $< B >$, respectively. Let us consider some examples on using this language to describe spreadsheet templates.

For example, template $Dept - < string > \mid ID - < int > \mid Name - < string > \mid Address - < string >$ describes a class of tabular spreadsheet instances with four fixed columns, each of which consists of a description label (i.e., header, such as **Dept**, **ID**, **Name**, **Address**) at the top and a set of subsequent text/numeric data cells (represented by types string, int,...). The spreadsheet in Figure 4.1(a) is an instance of this template.

Template $\langle \text{Dept} - \text{ID} - \text{Name} - \varepsilon - \varepsilon \rangle \mid \langle \text{string} - \text{int} - \text{string} - \varepsilon - \varepsilon \rangle$ describes a class of spreadsheet instances with two fixed columns; the first column contains groups of labels, each of which contains three labels and two empty cells; the second column contains groups of data cells adjacent to labels, each of which contains three data cells and two empty cells. The spreadsheet in Figure 4.1(b) is an instance of this template. Similarly, the spreadsheet in Figure 4.1(c) is an instance of template $\langle \text{Dept} - \text{ID} - \text{Address} - \text{Name} - \varepsilon \rangle \mid \langle \text{string} - \text{int} - \text{string} - \text{string} - \varepsilon \rangle$.

4.3.2 Inferring Templates

Given a spreadsheet, it is desirable to infer a template that characterizes its structure. The problem is challenging since spreadsheets may not impose any restrictions on how to organize data over a tabular grid. In fact, the spreadsheet may not have enough information for inferring a template. Therefore, template inference is typically ambiguous and users generally need to provide input to resolve ambiguities during inference process [35].

Abraham et al. [35] present an inference technique based on the cells containing similar formulas to identify hex and vex groups (See Appendix A.1 for a detailed description). However, there are numerous spreadsheets in which formula cells are not available [77] (e.g., spreadsheets in Figures 4.1(a), 4.1(b), and 4.1(c)). As a result, in addition to that technique, we also provide an inference technique based on the common spreadsheet presentation patterns we proposed in [101], including the table, repeater, and hierarchical presentations. For instance, the inference algorithm for a repeater presentation in the vertical direction is shown in Algorithm 1. Recall that a vertical repeater presentation contains two columns, each of which contains instances of a vex group. Algorithm 1 extracts the two first instances, that characterize these vex groups, based on identifying the last empty row of the two instances. By using this algorithm, the template of the instance in Figure 4.1(b) is inferred as shown in Figure 4.2 in a new worksheet (table A1:B5), where the two vex groups with some default values are shaded light orange (vex groups A1:A5 and

B1:B5).

Input: Start and end coordinates of the instance: (x, y) and $(x + 1, y')$

Output: Start and end coordinates of the inferred template

begin

$i \leftarrow 0$;

repeat

$i \leftarrow i + 1$;

until $(x, y + i) = \varepsilon$ and $(x, y + i + 1) \neq \varepsilon$;

return (x, y) and $(x + 1, y + i)$

end

Algorithm 1: Template inference for a vertical repeater presentation.

To generate a string-based representation of an inferred template (e.g., the template in Figure 4.2), the following main steps are performed:

- First, stand at the start coordinate (i.e., top-left cell) of the inferred template (e.g., cell A1 in Figure 4.2).
- Then, traverse all columns of the template from left to right. If meet a hex group, generate a pair of (“[”, “]”) and put the result of traversing each column in the hex group in this pair. Between two columns (e.g., columns A1:A5 and B1:B5 in Figure 4.2), a column and a hex group, or two hex groups, generate “.”.
- For each column, traverse all cells of the column from top to bottom. If meet a vex group, then generate a pair of (“<”, “>”) and put the result of traversing each cell in the vex group in this pair. Between two cells (e.g., cells A1 and A2 in Figure 4.2), a cell and a vex group, or two vex groups, generate “—”.
- For each cell : (i) If the cell contains a formula f , generate “ f ”; (ii) If the cell is an empty cell, generate “ ε ”; (iii) If the cell contains a label l , generate “ l ”; (iv) If the cell is a data cell, generate its type. To decide whether a cell inside a hex group/vex group is a data cell or a label, we rely on the observation: If the cell is referenced by a formula or values of the cell are changed in instances of the hex group/vex group in the original spreadsheet (i.e., the spreadsheet

| | A | B | C | D |
|---|------|------|---|---|
| 1 | Dept | IT | | |
| 2 | ID | 1111 | | |
| 3 | Name | Ann | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |

Figure 4.2: Inferred template of the instance in Figure 4.1(b) in a new worksheet

from which the template is inferred), it is a data cell; otherwise, it is a label. For example, in Figure 4.2, cell B1 is a data cell (since its values are changed in vex group instances in the original spreadsheet in Figure 4.1(b)), while cell A1 is a label. For a cell outside vex groups/hex groups, if the cell is referenced by a formula, it is a data cell; otherwise, it is a label.

Note that each generated token is separated from the other tokens by a single whitespace that is convenient for tokenizing later. By applying the above steps, the string-based representation of the template in Figure 4.2 is generated as “< Dept – ID – Name – ε – ε > | < string – int – string – ε – ε >”. This string is then shown up to the user for validation and editing.

4.4 Reuse Recommendation Algorithm

There are numerous similarity functions for measuring similarities, but no single function is known to be the best one, basically depending on the application domain. There are mainly two relevant approaches, namely *character-based* and *token-based* similarity metrics [14]. Character-based approach relies on the notion of *edit distance* which measures the minimum number of edit operations needed to transform one string to the other. Edit operation is an insertion, deletion, or substitution of a single character.

For example, the edit distance between “microsoft” and “mcrosoft” is 1, with one delete operation. Edit distance captures well typographical errors (words with

alternative spellings). However, when there is a rearrangement of words in a string, character-based metrics fail to capture the similarity (e.g., “Bill Gates” and “Gates, Bill”). Token-based approach is designed to solve this problem, in which strings are tokenized according to word boundaries and popular *set-based* similarity measures (e.g., Jaccard and Cosine) are then used to compute similarity.

Regarding algorithm illustration, we specifically focus on the Jaccard similarity, a commonly used function for defining similarity between sets. It has been shown that supporting Jaccard similarity efficiently leads to sound implementations of other similarity functions (e.g., edit distance and Cosine similarities) [43, 57, 141]. As mentioned earlier (Section 4.2), we find in Γ_T all templates that are similar to P_J , instead of looking for all similar template pairs in the set of templates $\Gamma_T \cup \{P_J\}$ as in the case of similarity join. Therefore, our algorithm is designed based on modifying the recently proposed algorithm PPJoin+ [141, 103], which has been shown to outperform previous ones on similarity join. The main steps of the algorithm are sketched in Algorithm 2.

We will intuitively illustrate these steps in via the motivating example. Suppose that Γ_T already contains the mappings of the instance I_0 (Figure 4.1(a)) and the instance I_1 (Figure 4.1(b)) to the target schema (Figure 4.1(d)). The specified template-level mapping formulas are:

- $\text{Emp} = \text{A2:D}(\text{value}=\text{empty})$ (for I_0)
- $\text{Dept} = \text{B1:B}(\text{next}=\text{bottom}(\text{Dept})+5)$ (for I_1)
- $\text{EmployeeID} = \text{B2:B}(\text{next}=\text{bottom}(\text{EmployeeID})+5)$ (for I_1)
- $\text{EmployeeName} = \text{B3:B}(\text{next}=\text{bottom}(\text{EmployeeName})+5)$ (for I_1).

The user now wants to map instance J in Figure 4.1(c) to the target schema.

The templates of the three instances are described in Section 4.3: $P_{I_0} = \text{“Dept} - \langle \text{string} \rangle \mid \text{ID} - \langle \text{int} \rangle \mid \text{Name} - \langle \text{string} \rangle \mid \text{Address} - \langle \text{string} \rangle \text{”}$, $P_{I_1} = \text{“} \langle \text{Dept} - \text{ID} - \text{Name} - \varepsilon - \varepsilon \rangle \mid \langle \text{string} - \text{int} - \text{string} - \varepsilon - \varepsilon \rangle \text{”}$, and $P_J = \text{“} \langle \text{Dept} - \text{ID} - \text{Address} - \text{Name} - \varepsilon \rangle \mid \langle \text{string} - \text{int} - \text{string} - \text{string} - \varepsilon \rangle \text{”}$. The threshold to be set is 0.8: $\theta = 0.8$.

Input: A collection of specified mappings Γ_T ;
 A new instance J conforming to template P_J ;
 Jaccard similarity function f and threshold θ
Output: All pairs (P_I, P_J) , $P_I \in \Gamma_T$ such that $f(P_I, P_J) \geq \theta$
begin
 1. Tokenize templates in Γ_T and template P_J into sets of tokens (i.e., records).
 2. Records are canonicalized according to the document frequency ordering \mathcal{O}_{df}
 3. Create inverted lists on tokens that appear in templates in Γ_T and generate candidates for P_J by probing these inverted lists.
 4. Reduce candidate size for P_J using size filtering and positional filtering.
 5. Verify final candidates using similarity function f such that $f \geq \theta$.
end

Algorithm 2: The algorithm for reuse recommendation.

At first step, we transform each template into a set of tokens according to the delimiter whitespace. Since tokens may occur multiple times in a string, we will convert a multiset of tokens into a set of tokens by treating each subsequent occurrence of the same token as a new token [57]. Such a set of tokens is called a *record*. For example, $P_{I_0} = \{\text{Dept}_0, -_0, <_0, \text{string}_0, >_0, \text{ID}_0, -_1, <_1, \text{int}_0, >_1, \text{Name}_0, -_2, <_2, \text{string}_1, >_2, \text{Address}_0, -_3, <_3, \text{string}_2, >_3, P_{I_1} = \{<_0, \text{Dept}_0, -_0, \text{ID}_0, -_1, \text{Name}_0, -_2, \varepsilon_0, -_3, \varepsilon_1, >_0, \text{string}_0, -_4, \text{int}_0, -_5, \text{string}_1, -_6, \varepsilon_2, -_7, \varepsilon_3, >_1\}$ and $P_J = \{<_0, \text{Dept}_0, -_0, \text{ID}_0, -_1, \text{Address}_0, -_2, \text{Name}_0, -_3, \varepsilon_0, >_0, \text{string}_0, -_4, \text{int}_0, -_5, \text{string}_1, -_6, \text{string}_2, -_7, \varepsilon_1, >_1\}$.

In the second step, to compare records, a record is *canonicalized* by sorting its tokens according to a certain global ordering \mathcal{O} defined on the token universe \mathcal{U} . The document frequency of a token is the number of records containing the token. A document frequency ordering \mathcal{O}_{df} arranges the tokens of a record according to the increasing order of tokens' document frequencies. \mathcal{O}_{df} favors rare tokens in prefixes and hence produces a small candidate size as presented in next steps. For example, the token universe, the tokens' document frequencies, and token's orders of P_{I_0} , P_{I_1} , and P_J are presented in Tables 4.1 and 4.2 (DF is the abbreviation for "Document Frequency").

| Token | Dept ₀ | - ₀ | < ₀ | string ₀ | > ₀ | l ₀ | ID ₀ | - ₁ | < ₁ | int ₀ | > ₁ | l ₁ | Name ₀ | - ₂ | < ₂ |
|-------|-------------------|----------------|----------------|---------------------|----------------|----------------|-----------------|----------------|----------------|------------------|----------------|----------------|-------------------|----------------|----------------|
| DF | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 3 | 3 | 1 |
| Order | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 1 | 28 | 29 | 2 |

Table 4.1: Tokens' document frequencies and token's orders of P_{I_0} , P_{I_1} , and P_J (Part 1)

| Token | string ₁ | > ₂ | l ₂ | Address ₀ | - ₃ | < ₃ | string ₂ | > ₃ | - ₄ | - ₅ | - ₆ | - ₇ | ε_0 | ε_1 | ε_2 | ε_3 |
|-------|---------------------|----------------|----------------|----------------------|----------------|----------------|---------------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|
| DF | 3 | 1 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 |
| Order | 30 | 3 | 4 | 9 | 31 | 5 | 10 | 6 | 13 | 14 | 15 | 16 | 11 | 12 | 7 | 8 |

Table 4.2: Tokens' document frequencies and token's orders of P_{I_0} , P_{I_1} , and P_J (Part 2)

Consequently, P_{I_0} , P_{I_1} , and P_J are canonicalized according to \mathcal{O}_{df} with the following results: $P_{I_0} = \{l_1, <_2, >_2, l_2, <_3, >_3, \text{Address}_0, \text{string}_2, \text{Dept}_0, -_0, <_0, \text{string}_0, >_0, l_0, \text{ID}_0, -_1, <_1, \text{int}_0, >_1, \text{Name}_0, -_2, \text{string}_1, -_3\}$, $P_{I_1} = \{\varepsilon_2, \varepsilon_3, \varepsilon_0, \varepsilon_1, -_4, -_5, -_6, -_7, \text{Dept}_0, -_0, <_0, \text{string}_0, >_0, l_0, \text{ID}_0, -_1, <_1, \text{int}_0, >_1, \text{Name}_0, -_2, \text{string}_1, -_3\}$, and $P_J = \{\text{Address}_0, \text{string}_2, \varepsilon_0, \varepsilon_1, -_4, -_5, -_6, -_7, \text{Dept}_0, -_0, <_0, \text{string}_0, >_0, l_0, \text{ID}_0, -_1, <_1, \text{int}_0, >_1, \text{Name}_0, -_2, \text{string}_1, -_3\}$.

In the third step, given two canonicalized records x and y with $O(x, y) = |x \cap y|$, by using the transformation $|x \cup y| = |x| + |y| - |x \cap y|$, we have:

$$Jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|} \geq \theta \Leftrightarrow O(x, y) \geq \alpha = \frac{\theta}{\theta + 1} * (|x| + |y|) \quad (4.1)$$

$$Jaccard(x, y) \geq \theta \Rightarrow \theta * |x| \leq |y|, \theta * |y| \leq |x| \quad (4.2)$$

Let p -*prefix* of a record x be the first p tokens of x . *Prefix-filtering principle* [57] states that $(|x| - \alpha + 1)$ -prefix of x and $(|y| - \alpha + 1)$ -prefix of y must share at least one token if (x, y) are a candidate pair. The prefix-filtering principle is used in building *inverted lists* (i.e., inverted indices). In terms of information retrieval, input template P_J can be seen as a query and the templates in Γ_T can be considered as the set of matching documents. The key idea is that we create inverted lists for each token of existing templates in Γ_T and we index prefixes with certain lengths, instead of the whole records. Obviously, if two records (x, y) are a candidate pair, then they share at least one token. Given a candidate pair (x, y) , the prefix of

length $|x| - \lceil \theta * |x| \rceil + 1$ of x and the prefix of length $|y| - \lceil \theta * |y| \rceil + 1$ of y must share at least one token. This is because $|x| - \lceil \theta * |x| \rceil + 1 \geq |x| - \alpha + 1$ and $|y| - \lceil \theta * |y| \rceil + 1 \geq |y| - \alpha + 1$ (based on Equation 4.2).

Therefore, given a template P_I in Γ_T , we only need to index a prefix of length $|P_I| - \lceil \theta * |P_I| \rceil + 1$. We index both tokens and their positions in the prefixes so that positional filtering can be applied later. Then, with respect to the input template P_J , we match each token of the $(|P_J| - \lceil \theta * |P_J| \rceil + 1)$ -prefix of P_J against the inverted lists and generate candidates for P_J by union the matched lists.

For example, the 5-prefix of P_{I_0} to be indexed is $\{l_1, <_2, >_2, l_2, <_3\}$ with the following lists: $\text{list}(l_1) = \{(P_{I_0}, 1)\}$; $\text{list}(<_2) = \{(P_{I_0}, 2)\}$; $\text{list}(>_2) = \{(P_{I_0}, 3)\}$; $\text{list}(l_2) = \{(P_{I_0}, 4)\}$; $\text{list}(<_3) = \{(P_{I_0}, 5)\}$. Also, 5-prefix of P_{I_1} to be indexed is $\{\varepsilon_2, \varepsilon_3, \varepsilon_0, \varepsilon_1, -_4\}$ with the following lists: $\text{list}(\varepsilon_2) = \{(P_{I_1}, 1)\}$; $\text{list}(\varepsilon_3) = \{(P_{I_1}, 2)\}$; $\text{list}(\varepsilon_0) = \{(P_{I_1}, 3)\}$; $\text{list}(\varepsilon_1) = \{(P_{I_1}, 4)\}$; $\text{list}(-_4) = \{(P_{I_1}, 5)\}$. Next, each token in the 5-prefix of $P_J = \{\text{Address}_0, \text{string}_2, \varepsilon_0, \varepsilon_1, -_4\}$ is matched against the above lists. Finally, the candidate set is created by applying the union operation on the matched lists: $\text{list}(\varepsilon_0) \cup \text{list}(\varepsilon_1) \cup \text{list}(-_4) = \{(P_{I_1}, 3), (P_{I_1}, 4), (P_{I_1}, 5)\}$. As can be seen, P_{I_0} is filtered out and only P_{I_1} is passed to the next step.

In the fourth step, given a candidate pair (x, y) , we reduce the candidate size based on size filtering (see Equation 4.2) and positional filtering. The positional filtering is stated as follows:

- Let token $\omega = x[i]$, ω partitions the record x into the left partition $x_l(\omega) = x[1..i]$ and the right partition $x_r(\omega) = x[i + 1..|x|]$. If $O(x, y) \geq \alpha$, then for every token $\omega \in x \cap y$, $O(x_l(\omega), y_l(\omega)) + \min(|x_r(\omega)|, |y_r(\omega)|) \geq \alpha$.

We have $|P_{I_1}| = |P_J| = 23$ so size filtering is satisfied. For the common token ' ε_0 ', $O(P_{I_1l}(\varepsilon_0), P_{Jl}(\varepsilon_0)) + \min(|P_{I_1r}(\varepsilon_0)|, |P_{Jr}(\varepsilon_0)|) = 1 + \min(20, 20) = 21 > \alpha = 20.44$. Similarly, positional filtering is also valid for the other common tokens ' ε_1 ' and ' $-_4$ ' since:

- $O(P_{I_1l}(\varepsilon_1), P_{Jl}(\varepsilon_1)) + \min(|P_{I_1r}(\varepsilon_1)|, |P_{Jr}(\varepsilon_1)|) = 2 + \min(19, 19) = 21 > \alpha =$

20.44

- $O(P_{I_l}(-4), P_{J_l}(-4)) + \min(|P_{I_r}(-4)|, |P_{J_r}(-4)|) = 3 + \min(18, 18) = 21 > \alpha = 20.44$

Hence, the pair (P_{I_1}, P_J) can be passed to the final step for verification. Note that for really large datasets, *suffix filtering* [141] can also be used to prune more candidates. This filtering method is a generalization of the positional filtering to the suffixes of the records by converting the overlap constraint to the equivalent Hamming distance constraint.

Our example has $|P_{I_1}| = |P_J| = 23$ so size filtering is satisfied. For the common token ' ε_0 ', $O(P_{I_{1l}}(\varepsilon_0), P_{Jl}(\varepsilon_0)) + \min(|P_{I_{1r}}(\varepsilon_0)|, |P_{Jr}(\varepsilon_0)|) = 1 + \min(20, 20) = 21 > \alpha = 20.44$. Positional filtering is also valid for the other common tokens ' ε_1 ' and ' -4 ' because: $\min(|P_{I_{1r}}(\varepsilon_1)|, |P_{Jr}(\varepsilon_1)|) = 2 + \min(19, 19) = 21$ and $\min(|P_{I_{1r}}(-4)|, |P_{Jr}(-4)|) = 3 + \min(18, 18) = 21$. As a result, the pair (P_{I_1}, P_J) can be passed to the final step for verification.

In the final step, Jaccard similarity function is computed to verify the candidate pair (P_{I_1}, P_J) . Recall that threshold θ can be equivalently converted to the overlap threshold α according to Equation (4.1). $Jaccard(P_{I_1}, P_J) = \frac{|P_{I_1} \cap P_J|}{|P_{I_1} \cup P_J|} = \frac{21}{25} = 0.84 > \theta \Leftrightarrow O(P_{I_1}, P_J) = 21 > \alpha$. Thus, M_{I_1} can be potentially reused for M_J . In particular, the two following mapping formulas are reused completely:

- `Dept = B1:B(next=bottom(Dept)+5)`
- `EmployeeID = B2:B(next=bottom(EmployeeID)+5).`

The mapping formula `EmployeeName = B3:B(next=bottom(EmployeeName)+5)` needs to be modified slightly to `EmployeeName = B4:B(next=bottom(EmployeeName)+5)`. In addition to that, the user needs to write a new mapping formula for the optional label `Address`: `Address = B3:B(next=bottom(Address)+5)`. M_J is then stored in the mapping repository for future reuse.

It is worth noting that the *start coordinate* (i.e, top-left cell) of an instance is important in reuse. In the above example, starting coordinates of P_{I_1} and P_J are both

(1, 1) so we do not need to change mapping formulas in a recommendation. Suppose that the starting coordinate of P_J is now (2,2), then mapping formulas of P_{I_1} should be offset based on the row and column differences between two start coordinates. For example, two mappings that can be reused completely are offset as follows: $Dept = C2:C(next=bottom(Dept)+5)$ and $ID = C3:C(next=bottom(ID)+5)$.

The algorithm we presented above can be easily switched to other similarity metrics [141]. In the case of cosine similarity, the length of prefix to be indexed for a string x is $|x| - [\theta * |x|] + 1$; the size filtering threshold is $[\theta^2 * |x|]$; overlap threshold for positional filtering is $\alpha = [\theta * \sqrt{|x| * |y|}]$. The algorithm is also applicable to the edit distance with threshold δ if we tokenize strings to *q-grams*. The necessary condition for two strings (x, y) satisfying the threshold δ is their corresponding q-gram sets must have overlap no less than $\alpha = (max(|x|, |y|) + q - 1) - q * \delta$ [81]. Hence, prefix to be indexed for a string x is $q * \delta + 1$; the filtering size threshold is $|x| - \delta$; threshold for positional filtering is $x - q * \delta$.

4.5 Implementation

4.5.1 Architecture

Figure 4.3 extends the architecture of TranSheet presented in Chapter 3 with the following main components: (i) Template inference engine infers the template of an existing spreadsheet and generates the corresponding string (Section 4.3); (ii) Mapping repository stores specified mapping information (presented below); (iii) Reuse recommendation engine uses information stored in the mapping repository and recommends specified mapping formulas for reuse (Section 4.4).

4.5.2 Mapping Repository Organization

The main tables that form the basis for designing the mapping repository are as follows:

- Mappings(MId, InstanceId, TemplateId, SchemaId)

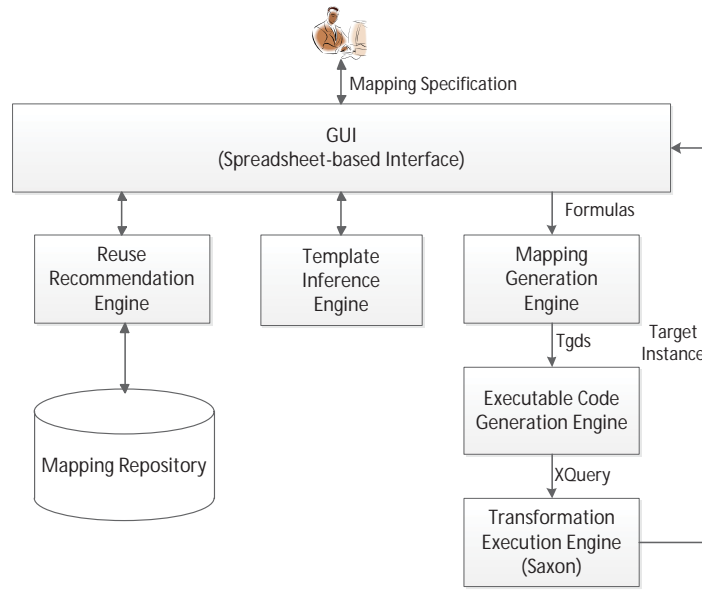


Figure 4.3: TranSheet architecture for spreadsheet-based data transformation reuse

- MappingFormulas(MEId, MId, EId, MappingFormula)
- Schemas(SchemaId, SchemaName)
- Elements(EId, Name, Type, Parent, SchemaId)
- Instances(InstanceId, StartCoordinate, EndCoordinate)
- Cells(CellId, Coordinate, Value, Formula, InstanceId)
- Templates(TemplateId, Representation)

Table **Mappings** encodes mappings from spreadsheet instances to target schemas where attributes **MId**, **InstanceId**, **TemplateId**, and **SchemaId** are a mapping identifier, an instance identifier, a template identifier, and a target schema identifier, respectively. Each mapping in table **Mappings** consists of a set of mapping formulas stored in table **MappingFormulas**, in which attributes **MEId**, **MId**, **EId**, and **MappingFormula** are a mapping formula identifier, a mapping identifier, a schema element identifier, and a mapping formula, respectively.

Tables **Schemas** and **Elements** encode the target schemas and their corresponding elements, respectively. Table **Schemas** stores a schema identifier **SchemaId** and a

| Mappings | | | | Schemas | |
|----------|------------|------------|----------|----------|------------|
| MId | InstanceId | TemplateId | Schemald | Schemald | SchemaName |
| M0 | I0 | P0 | Sch0 | Sch0 | Employees |
| M1 | I1 | P1 | Sch0 | | |

| Elements | | | | | MappingFormulas | | | |
|----------|---------|------|--------|----------|-----------------|-----|-----|-----------------------------------|
| EId | Name | Type | Parent | Schemald | MEId | MId | EId | MappingFormula |
| E0 | Target | Rcd | - | Sch0 | ME0 | M0 | E1 | A2:D(value=empty) |
| E1 | Emp | Set | E0 | Sch0 | ME1 | M1 | E3 | B1:B(next=bottom(Dept)+5) |
| E2 | * | Rcd | E1 | Sch0 | ME2 | M1 | E4 | B2:B(next=bottom(EmployeeID)+5) |
| E3 | Dept | Str | E2 | Sch0 | ME3 | M1 | E5 | B3:B(next=bottom(EmployeeName)+5) |
| E4 | ID | Int | E2 | Sch0 | | | | |
| E5 | Name | Str | E2 | Sch0 | | | | |
| E6 | Address | Str | E2 | Sch0 | | | | |

| Templates | | Instances | |
|------------|---|------------|------------------|
| TemplateId | Representation | InstanceId | Start Coordinate |
| P0 | Dept - < string > - ID - < int > Name... | I0 | A1 |
| P1 | < Dept - ID - Name - E - E > < string - ... | I1 | A1 |

| Cells | | | | |
|--------|------------|---------|---------|------------|
| CellId | Coordinate | Value | Formula | InstanceId |
| C0 | A1 | Dept | Dept | I0 |
| C1 | B1 | ID | ID | I0 |
| C2 | C1 | Name | Name | I0 |
| C3 | D1 | Address | Address | I0 |

Figure 4.4: Mapping Repository Organization

schema name **SchemaName**. Table **Elements** represents the graph representation of a target schema (Section 3.2). Each tuple of this table corresponds to a node of the graph where attributes **Name**, **Type**, **Parent**, and **SchemaID** specify a node's label, a node's type, a node's parent, and an identifier of a target schema, respectively.

Tables **Instances** and **Cells** are used to model a spreadsheet instance and its non-empty cells. Table **Templates** stores a template identifier **TemplateId** and its corresponding string representation **Representation**. For example, tables in Figure 4.4 are created for mapping instances I_0 in Figure 4.1(a) and I_1 in Figure 4.1(b) to the target schema in Figure 4.1(d).

4.6 Evaluation

4.6.1 Performance

Experimental setup. We use the EUSES spreadsheet corpus [77], which consists of 4498 spreadsheets collected from various sources (e.g., teaching courses, personal

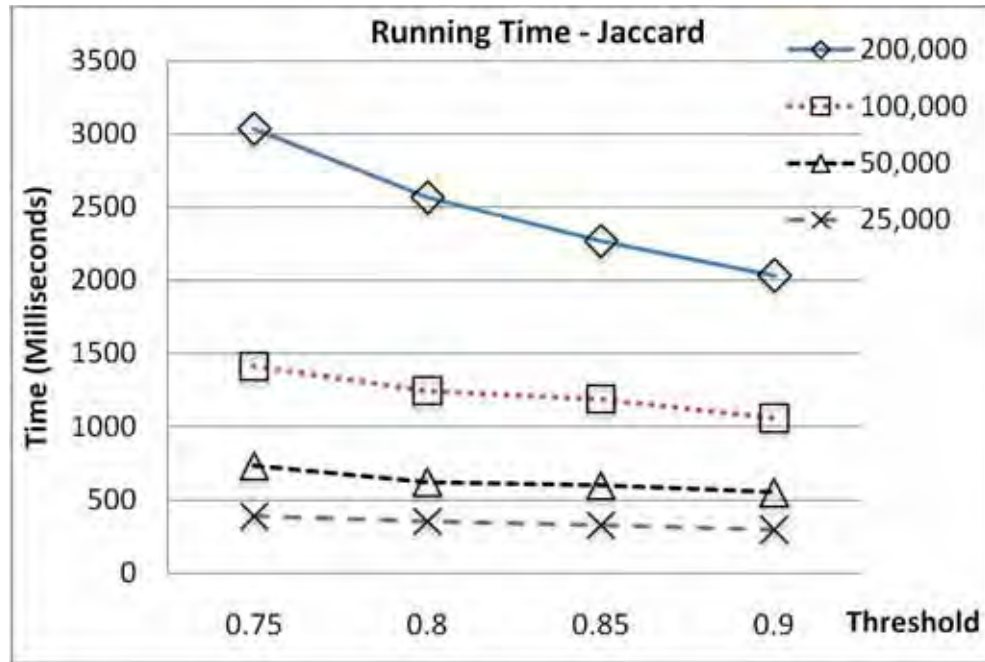


Figure 4.5: Performance graph of Jaccard similarity.

databases, and financial data), for evaluation. This corpus was widely used by many works on spreadsheet research (e.g., [35]). These spreadsheets have been used by numerous works on spreadsheet template inference [35].

We select 10 spreadsheets in the corpus whose templates can be inferred using techniques presented in Section 4.3. The average length of these templates is 104. For each spreadsheet S , the corresponding template $Temp_S$ is modified based on small incremental changes to generate 25000, 50000, 100000, and 200000 variants of it. The following operations are applied: (i) Insert a new column (row) inside a hex group (vex group) of $Temp_S$ at an arbitrary position; (ii) Insert a new column (row) outside the hex groups (vex groups) of $Temp_S$ at an arbitrary position; (iii) Delete an existing column (row) inside a hex group (vex group) of $Temp_S$; (iv) Delete an existing column (row) outside the hex groups (vex groups) of $Temp_S$. Then, apply these operations again to the newly generated templates and so on. All experiments were performed on a laptop with Intel Core 2 Duo 2.1GHz, 3GB RAM, and Windows Vista Home Premium SP2.

Methodology. Template $Temp_S$ of spreadsheet S is matched against its gen-

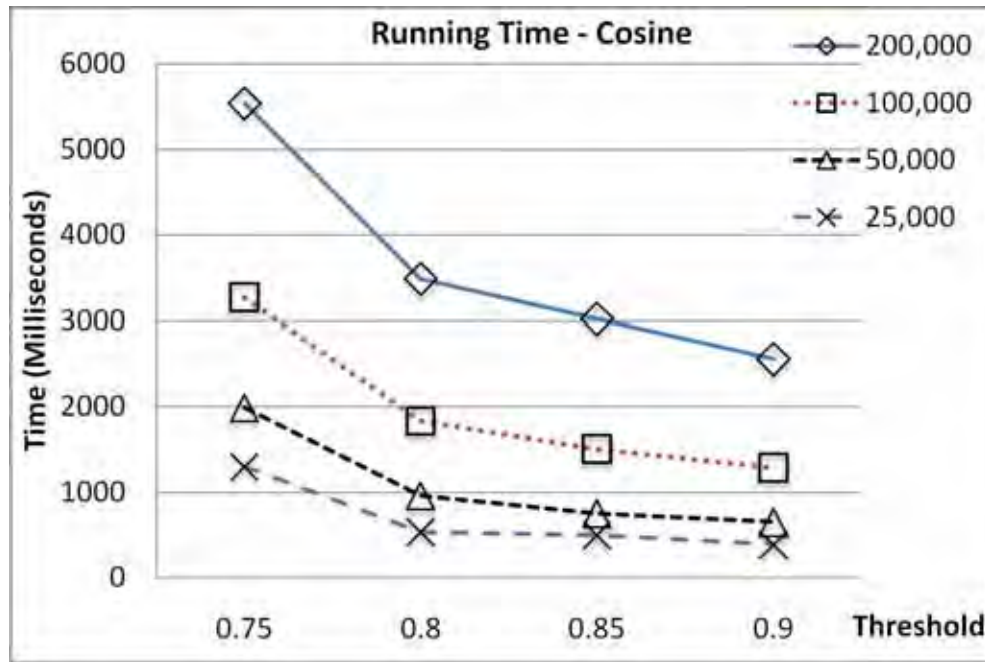


Figure 4.6: Performance graph of Cosine similarity.

erated variants stored in the mapping repository. We measure the running time of this matching for each fixed threshold. We use four thresholds, namely 0.9, 0.85, 0.8, and 0.75. Each experiment covers two similarity measures, namely Jaccard and Cosine. We then average the results of the 10 above selected spreadsheets.

Observations. The experimental results on performance are shown in Figures 4.5 and 4.6. There was no problem with the amount of available memory during experimentation (e.g., out of memory error). Consider the performance graph for Jaccard similarity in Figure 4.5. As can be seen, the running time increases when the threshold decreases and this trend is clearer when the number of variants is larger. This can be explained by two main reasons: (i) the number of inverted lists is larger for a small threshold so it takes more time to build and probe them; (ii) the candidate size increases for a smaller threshold. Basically, the running time grows almost linearly with the increase of the number of variants for each threshold.

Regarding Cosine similarity (Figure 4.6), the findings are essentially similar to those of Jaccard similarity. However, for each threshold, the running time is generally longer since the constraints of Cosine (e.g., length of the prefix, size filtering

threshold and overlap threshold) are looser than those of Jaccard. Furthermore, the running time increases considerably when changing from threshold 0.8 to 0.75 due to a surge of the candidate size at threshold 0.75, while it is a modest growth for other threshold changes, namely from 0.9 to 0.85 and from 0.85 to 0.8.

4.6.2 Effectiveness

Experimental setup. Since the size of the above repository (See Section 4.6.1) is quite big, to properly evaluate the reuse effectiveness, we create a smaller repository as the following. We select 10 spreadsheets from the EUSES corpus. For each spreadsheet, we represent it using table, repeater (separated by one blank column), and hierarchical (separated by one blank column) presentations. We then modify the table presentation by inserting a new column at an arbitrary position, deleting an existing column, or exchanging the orders of two existing columns. Afterwards, the modified table presentation is represented using corresponding repeater and hierarchical presentations with one separated blank column. Based on this procedure, for each spreadsheet S , we generate 50 variants of it. We also design the target schema T_S based on data of S . For each variant V_S of S , we manually write the mappings from V_S to T_S and save these mappings into the repository along with the template of V_S .

Methodology. To evaluate the effectiveness of our algorithm in the repository, we map S to T_S by reusing previously specified mappings of the variants of S stored in the repository, if any. We compared the real reusable mappings R (found manually) and the recommended mappings P found by our algorithm at four thresholds, namely 0.9, 0.8, 0.7, and 0.6. Let $I = R \cap P$, we use the quality measures employed by popular information retrieval studies [64, 107]:

- $precision = |I|/|P|$
- $recall = |I|/|R|$

Similar to the performance experiments, each experiment covers the Jaccard and Cosine similarities. The results of the 10 selected spreadsheets are averaged.

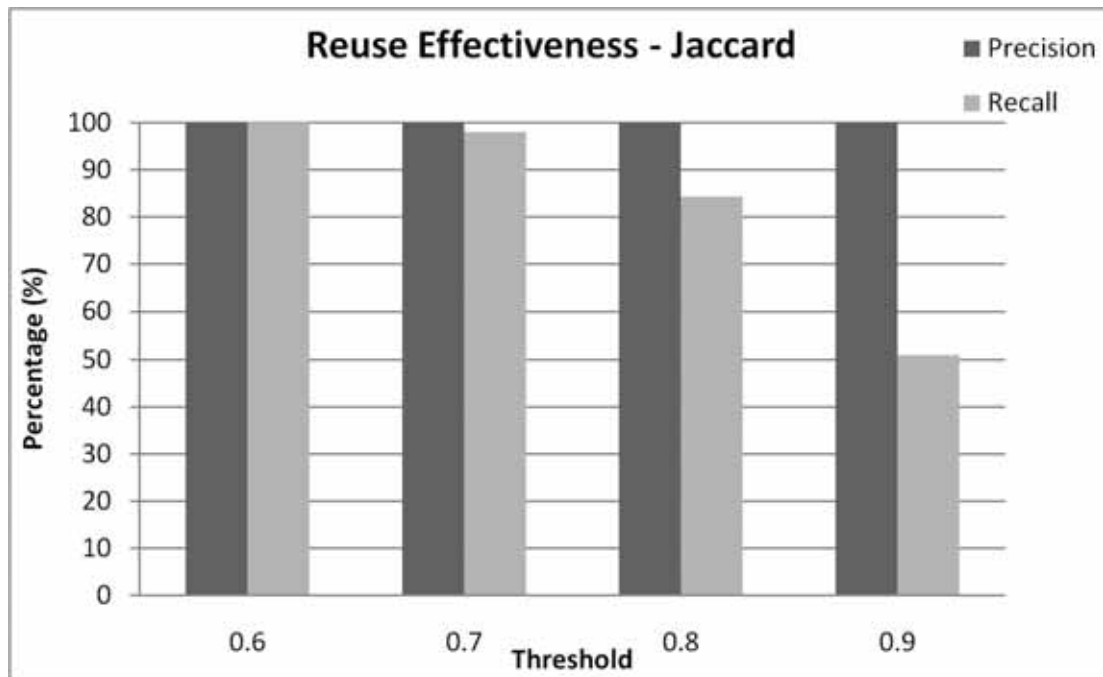


Figure 4.7: Reuse effectiveness of Jaccard similarity.

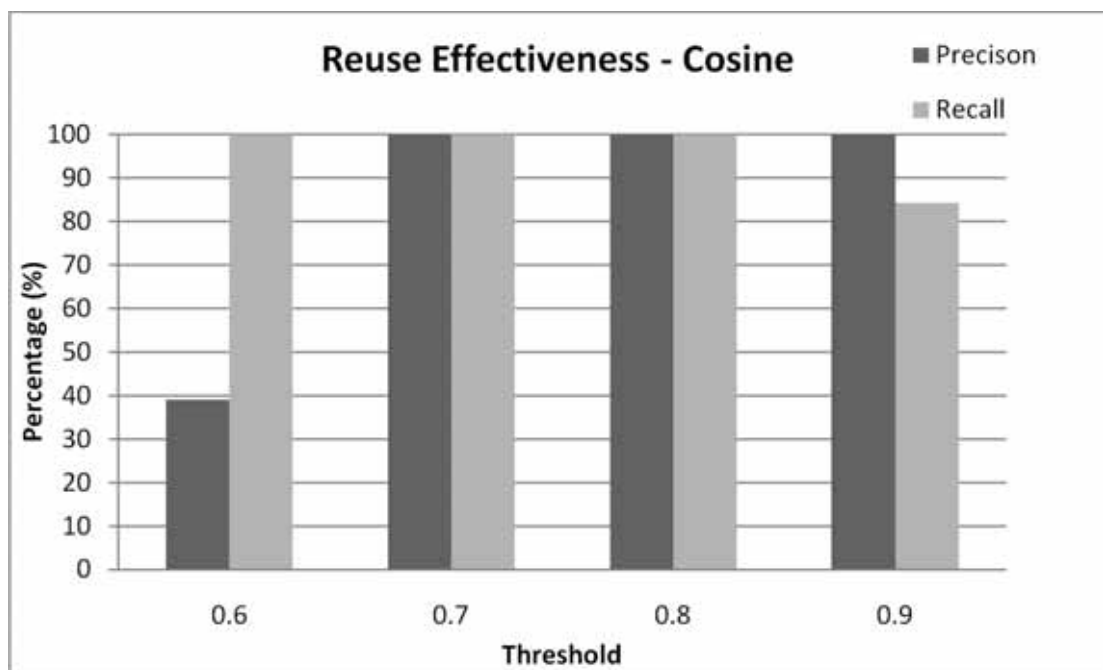


Figure 4.8: Reuse effectiveness of Cosine similarity.

Observations. The experimental results on reuse effectiveness are shown in Figures 4.7 and 4.8 for Jaccard and Cosine similarities, respectively. Consider the reuse effectiveness graph for Jaccard similarity shown in Figure 4.7. Precisions for all thresholds are all 100%. Meanwhile, the recall is higher when the threshold is smaller; at threshold 0.6, recall is 100%. This is because for a larger threshold, more reusable mappings are filtered out.

Since the constraints of Cosine similarity are looser, the reuse effectiveness results of Cosine (Figure 4.8) are slightly different from those of Jaccard. Precisions for all thresholds, except threshold 0.6 (recall is nearly 40%), are 100%. Some unusable recommended mappings appear at threshold 0.6. Except threshold 0.9 (recall is more than 80%), recalls for all thresholds are 100%.

4.7 Related Work

Reusing mapping information is first discussed in the survey of Rahm et al. [119] on schema matching (i.e., find semantic correspondences between the elements of two schemas). It is expected that in many cases schemas being matched can be very similar. Therefore, when matching different but similar schemas to the same destination schema (e.g., integrating new data sources into a data warehouse), it is possible to reuse existing mappings for entire schema structures, which results in significant savings of manual effort.

Inspired by that, COMA [64] proposes the MatchCompose operation for performing a join-like operation on a mapping path consisting of two or more mappings to deduce a new mapping (e.g., combine A-B, B-C, and C-D to derive a new mapping between A and D). COMA++ [44] extends COMA to deal with the cases where such mapping paths are unnecessary (i.e., one or multiple existing mappings can be reused for the given match problem) or not available (i.e., searching for mapping paths which are not available yet, but may be computed with less effort than a direct matching). To handle large and complex schemas, COMA++ also introduces the fragment-based matching approach, in which the source and target schemas are divided into fragments and each pair of source and target fragments are then

compared to detect the best matching pairs.

Madhavan et al. [107] leverage the knowledge in a corpus of schemas and mappings to improve schema matching results, in addition to the evidence that is available in the two schemas being matched. The corpus is used to augment the evidence about elements in the schemas being matched. Statistics gleaned from the corpus is used to infer domain constraints, which are crucial in achieving high matching accuracy. The reuse, however, is limited to element-to-element match, rather than larger concepts.

Recent work by Saha et al. [128] introduces the schema covering problem as a first step towards transformation reuse, which is an extension of Clio project [85, 79]. Given a complex schema, schema covering identifies a collection of common concepts (i.e., business objects) in a repository and creates a cover of the schema by these concepts. When a complex schema can be divided into smaller concepts, simple transformations defined among these concepts can be reused to define transformations among the complex schemas (e.g., by composing these simple transformations). This work assumes the existence of a concept repository and transformations among concepts. Building such repository is a challenging task including selection, cleaning, and unification of objects.

Mapping tool Altova MapForce [2] (MapForce for short) allows mapping an Excel 2007+ (XLSX) file to a target schema. This is done by first helping users specify the schema of the Excel file based on analyzing the file's OpenXML [125] format and then using the GUI of MapForce to specify mappings. Once users have finished defining mappings and processing functions, MapForce can automatically generate the program code (e.g., Java or C#) for transforming Excel data to the format required by an external application. The generated code can be reused for future mappings. However, this kind of reuse must be performed manually by users.

Our work focuses on reusing transformations from spreadsheet data to XML. We extend the spreadsheet-like formula mapping language we developed in our previous work [129] to solve this problem. With respect to the aforementioned reuse approaches, the main difference is that we allow users to specify mappings using spreadsheet-like formulas and then reuse previously specified mapping formulas for

a new spreadsheet instance that needs to be mapped to the target schema. Note that our work basically transforms data located on the tabular grid of a spreadsheet to XML, instead of extracting data from the spreadsheet, including macros and formulas. The recently proposed file format, namely OpenXML [125], allows external applications to easily extract data from Excel 2007 files (XLSX).

4.8 Summary

Transformation reuse is an important topic in information integration for both data integration and data exchange. In this paper, we considered the problem of reuse in transforming spreadsheet data to XML. We formulated the problem and proposed a solution based on the notions of spreadsheet template, mapping generalization, and similarity join. We extended the formula mapping language developed in our previous work. Given a spreadsheet instance, our algorithm recommended a list of previously specified mappings that can be reused for the instance. We implemented a prototype and evaluated the efficiency and effectiveness of the proposed solution via synthetic datasets. The experimental results confirmed the usefulness and viability of our approach. We believe this paper is an important step in building a simple and reusable data transformation framework (as outlined in the Clio project [128, 85]) in the context of spreadsheet-based data transformation.

Chapter 5

End-user oriented spreadsheet-based data transformation

In Chapter 3, we developed a spreadsheet-like formula language allowing users to specify mappings between spreadsheet data and the target schema. However, users still have to remember the complex syntax of the language to specify mappings, which may be challenging for spreadsheet novices. Furthermore, even expert users, who are already familiar with the language, also want to boost productivity by not having to write complex formulas from scratch. In this chapter, we study the problem of simplification in transforming spreadsheet data to structured formats. We propose a number of novel techniques that make spreadsheet-based data transformation available to non-technical users. First, we redesign the mapping interface of TranSheet based on nested tables. A schema matching module is also integrated to help users find correspondences between the source spreadsheet and the target schema. In addition, a collection of form-based operators are developed to help users graphically specify mappings, instead of remembering and writing complex formulas from scratch. Moreover, we provide a mechanism for automatically suggesting transformations from source columns to atomic target labels. We implement a prototype and conduct an extensive user study to evaluate the usability of our techniques.

5.1 Introduction

On one hand, powerful transformation languages (e.g., XSLT and XQuery) and visual mapping tools (e.g., Clio [85], Clip [117], +Spicy [112], Altova MapForce [1], Stylus Studio [9], MS BizTalk Mapper [3]) are largely developed with an enterprise setting which requires the expertise of professional programmers. On the other hand, data transformation has been increasingly necessary for non-technical users to analyze, manipulate and visualize data (e.g., social data analysis [137], Web mashup [139, 142], and SOA [39]).

In this chapter, we consider the problem of spreadsheet-based data transformation simplification, which makes spreadsheet-based data transformation available to non-technical users. In chapter 3, we developed a spreadsheet-like formula mapping language for specifying mappings between spreadsheet data and the target schema. But we mainly focus on the semantics and foundations of the language, rather than its usability aspects (e.g., interface design, graphical mapping specification, and transformation suggestion). In what follows, we review the state-of-the-art via a running example.

5.1.1 Running Example

Figure 5.1 depicts the running example for the whole chapter. Source spreadsheet containing order information grouped by order identifiers is shown in Figure 5.1(a). It must be mapped to the target schema shown in Figure 5.1(b).

The state-of-the-art helps users specify the schema of the source spreadsheet (e.g., using the layout specification language described in [104] or analyzing spreadsheet content [60]). Transformation can then be performed at schema level. One solution is to write programs using powerful transformation languages, such as XQuery and XSLT. However, this requires deep programming expertise, which is far beyond the capability of non-technical users. A more advanced solution is to use visual mapping tools to specify transformations via GUI. As described earlier in Section 2.5, current mapping tools mainly follow relationship-based metaphor [127]. More specif-



Figure 5.1: Running example: (a) Source spreadsheet; (b) Target schema

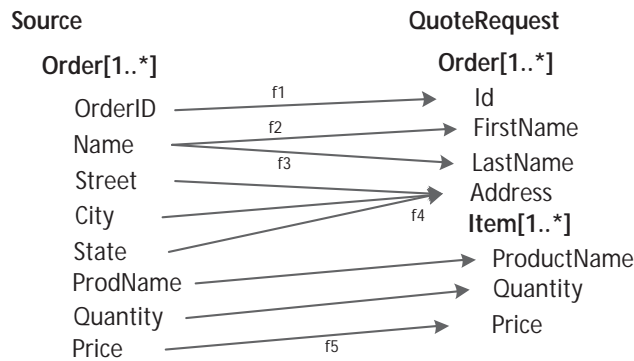


Figure 5.2: Mapping interface of a relationship-based mapping tool

ically, the source and target schemas are displayed on the left side and on the right side, respectively, of mapping interface. Correspondences are established by drawing lines connecting the source elements to the target elements. These lines may be annotated by one or more functions that are picked up from a built-in library to construct a more complex relationship. For instance, the mapping interface of a relationship-based mapping tool implementing the running example is depicted in Figure 5.2.

This flow-chart-like interface is typically cluttered and unintuitive as pointed out in the literature [126, 140] (See Section 1.1.2 for more details). In addition to that, users must familiarize themselves with the syntax and semantics of the unfamiliar functions of low-level programming languages, which vary from tool to tool. For

example, while the functions of Stylus Studio [9] are mainly identical to the ones of XSLT/XQuery and Java (e.g., FLWORS and IF blocks), MS BizTalk Mapper [3] provides a pre-defined library of .NET-powered functions. Finally, these mapping tools separate between two modes, namely *development mode* and *execution mode*. Like professional programming environments, user must compile a mapping specification to preview transformation result. This also raises another hassle because instant feedback is not available after each step of the mapping specification.

5.1.2 Contributions

In this chapter, we aim at addressing the aforementioned issues. More specifically, we make the following major contributions:

- The interface is redesigned based on nested tables, which is more intuitive and easy-to-use for non-technical users. With this new interface, users can preview entire transformation result right in the sheet containing the source data, which is convenient for side-by-side comparison, which is used to refine a mapping specification. Also, a matching module is integrated to help users semi-automatically find semantic correspondences between the source spreadsheet and the target schema (Section 5.2).
- A set of form-based transformation operators are proposed allowing users to specify mappings graphically. The benefits of these operators are two-fold: (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 in spreadsheet programming. We define these operators in a generic way in order to cover numerous transformation patterns. We provide a form customization mechanism allowing users to customize an existing operator to suit transformation needs. We also offer a history list allowing users to modify specified transformation operations (Section 5.3).
- TranSheet automatically suggests transformations from source columns to target labels. This relies on employing *Topes* [132], where each Tope is a category

of data with different formats and functions for transforming between these formats. Automated transformations are helpful when specifying transformations using formulas or form-based transformation operators is complicated and difficult. (Section 5.4).

- We implemented a prototype and conducted an extensive user study across a set of real spreadsheet-based transformation tasks to evaluate the usability of our proposed techniques. The experimental results show that TranSheet significantly reduces specification time and promotes users' satisfaction in comparison with state-of-the-art mapping tools (Sections 5.5 and 5.6).

The rest of this chapter is organized as follows. Section 5.2 describes how the mapping interface presented in Chapter 3 is redesigned to be more intuitive. The integration of a matching module is also discussed this section. We present form-based transformation operators that help users specify mappings graphically in Section 5.3. We rely on *Topes* [132] to automatically suggest transformations in Section 5.4. Section 5.5 presents the implementation details of a proof-of-concept prototype for the proposed techniques. Section 5.6 presents an extensive user study to evaluate the usability of TranSheet. We discuss related work in Section 5.7 and conclude in Section 5.8.

5.2 User Interface

Section 5.2.1 describes the new mapping interface of TranSheet. Section 5.2.2 presents the integration of a matching module into TranSheet.

5.2.1 Mapping Sheet

In Chapter 3, the interface of TranSheet is organized as follows (See Section 3.7). While the source spreadsheet is located inside the tabular grid of Excel on the left side, the target schema is located in the task pane of Excel on the right side (roughly similar to the organization depicted in Figure 5.1). However, users can only

preview some values associated with atomic labels, not the whole target instance. Additionally, the transformed target data are located outside the tabular grid, which is inconvenient for users to make side-by-side comparison between the source and target data.

The nested relational model [83] extends the relational model by relaxing the first normal assumption (i.e., A column can contain a nested table). This allows it to represent hierarchical data, which are commonly used in Web applications. Therefore, it has been widely used to represent semi-structured web data. Besides, the nested relational model is simple and intuitive for non-technical users to understand and manipulate. In the nested relational model, data is represented as nested tables.

We use nested tables to represent target data. In particular, the transformed target instance is a nested table, in which each column represents an atomic attribute or a sub-relation, each row represents a tuple. An atomic attribute can be one of the following types: *empty* (for representing an empty column), *integer*, *string*, *float*, and *datetime*. The benefits of using nested tables are two-fold: (i) the target instance is displayed inside the tabular grid; (ii) the whole target instance can be seen as the instant feedback of each mapping specification.

For example, in Figure 5.3, with respect to the TranSheet interface, the source spreadsheet data (Figure 5.3(a)) is displayed next to (separated by one blank column) the target nested table (Figure 5.3(b)) in a worksheet, namely *mapping sheet* (i.e., the sheet for specify mappings between the source spreadsheet and the target schema).

As can be observed in Figure 5.3(b), relation **Order** consists of 4 attributes **Id**, **FirstName**, **LastName**, **Address**, and one sub-relation **Item**. Sub-relation **Item**, in turn, has 3 attributes **ProductName**, **Quantity**, and **Price**.

In order to specify a mapping, the user selects a target label associated with an attribute or a sub-relation, and enters a corresponding mapping formula in the formula textbox. For example, in Figure 5.3, to specify the mapping **Order** = A2:H50, the user selects cell J1 containing the target label **Order** and inputs formula A2:H50

| | 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 | Twel | 5 | 2 |
| 4 | 42 | Ford Prefect | Addison | Sydney | NSW | Fish | 10 | 4 |
| 5 | 525 | Arthur Dent | Evans | Melbourne | VIC | Twel | 3 | 2 |
| 6 | 525 | Arthur Dent | Evans | Melbourne | VIC | Sea Bags | 5 | 2 |
| 7 | | | | | | | | |

(a)

| | I | K | L | M | N | O | P | Q |
|---|-------|-----------|----------|---------------------|-------------|----------|-------|---|
| 1 | Order | | | | | | | |
| 2 | Id | FirstName | LastName | Address | ProductName | Quantity | Price | |
| 3 | 42 | Ford | Prefect | Addison Sydney NSW | Beer | 4 | 3 | |
| 4 | | | | | Twel | 5 | 2 | |
| 5 | | | | | Fish | 10 | 4 | |
| 6 | 525 | Arthur | Dent | Evans Melbourne VIC | Twel | 3 | 2 | |
| 7 | | | | | Sea Bags | 5 | 2 | |

(b)

Figure 5.3: Side-by-side comparison: (a) Source spreadsheet data; (b) Target data represented using a nested table

in the formula textbox. Instant feedback for the mapping is provided from row 4 onwards shown in Figure 5.3(b).

5.2.2 Matching Sheet

Before starting specifying mappings, it is needed to identify semantic correspondences between source columns and target labels. Numerous schema matching systems has been developed to deal with this task, but they are mostly not readily public for integration [119]. Given the variety of available matching systems, our system is designed to flexibly integrate the outputs of them. Currently, we employ the matching system COMA++ [44], since it is publically available with multiple built-in matchers (See Section 2.3 for a detailed description on matchers) and its matching result can be reused.

Matching sheet is a worksheet designed to display matching result outputted from COMA++ and let users refine matching result. It is displayed as a two-column table, where each of the table is a mapping element: the left column consists of one target label and the right column consists of one or more source labels.

Suppose that the source spreadsheet in Figure 5.3(a) is represented by relation `Order(OrderId, Name, Street, City, State, ProdName, Quantity, Price)` (the names of the attributes are identical to the ones of the column headers in Figure 5.3(a)). The matching sheet for the running example is shown in Table 5.1.

To specify a mapping for the mapping element of each row in the matching

| | A | B | C | D | E |
|---|---|--------------------|---------------------|---|---|
| 1 | | Target | Source | | |
| 2 | | <u>Id</u> | OrderId | | |
| 3 | | <u>FirstName</u> | Name | | |
| 4 | | <u>LastName</u> | Name | | |
| 5 | | <u>Address</u> | Street, City, State | | |
| 6 | | <u>ProductName</u> | ProdName | | |
| 7 | | <u>Quantity</u> | Quantity | | |
| 8 | | <u>Price</u> | Price | | |

Table 5.1: Matching sheet for the running example

sheet, the user clicks on the link under a target label, and the user is redirected to the mapping sheet, where corresponding source columns and the target label are highlighted. The user can then enter a mapping formula in the formula textbox to specify a relationship between these columns and the target label. In this way, there is a smooth connection between the matching sheet and the mapping sheet.

For example, the user clicks on the link under label **Address** (in cell B5) of the matching sheet in Table 5.1. Then columns C, D, E, and cell M2 containing label **Address** are highlighted in the mapping sheet (Figure 5.3). After that, mapping **Address** =concatenate(C2:C50, “ ”, D2:D50, “ ”, E2:E50) can be specified.

5.3 Form-based Transformation Operators

In this section, we present a set of visual transformation operators that complements the formula mapping language presented in Chapter 3. They combine the power of graphical visualization and transformation patterns. Each operator characterizes a transformation pattern and is represented as a customizable form. A transformation operator is usually activated from a *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 target label and source columns being selected are shown up to the user.

Section 5.3.1 defines the structure of a transformation operator in a generic way. Section 5.3.2 provides a form customization mechanism allowing users to customize

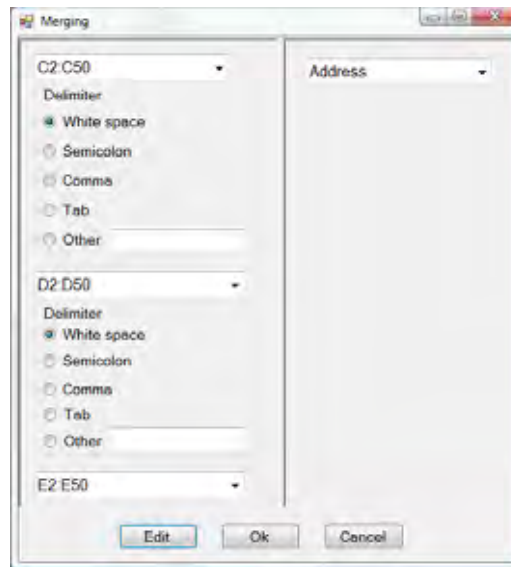


Figure 5.4: Merging Operator

a operator (activated from a contextual menu) to suit transformation needs. Section 5.3.3 presents the semantics and interface design of specific transformation operators for both value and structural mappings. Section 5.3.4 presents a history list allowing users to modify specified transformation operations. Section 5.3.5 describes the expressiveness of transformations supported by the operators of TranSheet.

5.3.1 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 of the form contains source components, the right side of the form consists of target components. An operator 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 (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 5.4 indicates one form-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 5.4 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 5.3.1 *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 $\{g_1, \dots, g_n\}$ where:*

- g_i is a form-element or a form-group, $i \in \{1, \dots, n\}$
- 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.

For example, while the source panel of the **Merging** operator depicted in Figure 5.4 consists of the set $\{C2:C50, \text{Delimiter}, D2:D50, \text{Delimiter}, E2:E50\}$ 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**.

5.3.2 Transformation Operator Customization

In this section, we present a set of available form customization operators offered by TranSheet. We first describe them in an abstract manner and then 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 this panel. 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:

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

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

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

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

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

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 :

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

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 :

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

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 :

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

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 :

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

Customization Operator Generation

With any form as a starting point, the user can edit it using a form editor in multiple iterations until the desired form is obtained. The *editing mode* of a form provides button-activated operations to modify the form. For instance, to enter the editing mode of the form shown in Figure 5.5(b), the user clicks on button **Edit** and a form editor is shown accordingly in Figure 5.5(a). To complete customization, the user clicks on button **Submit** (See Figure 5.5(a)). To reset a form editor to the original state, the user clicks on button **Reset** (See Figure 5.5(a)). In the following, we describe how each customization operator is realized.



Figure 5.5: (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 5.5(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 the Filtering operator shown in Figure 5.5(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 where a insertion button is activated.

Form-element/Form-group Deletion Form-elements or form-groups of a form that are irrelevant for transformation can be removed from the form by clicking on remove button marked “X” (Figure 5.5(a)). In the case of form-group deletion, if a group is not empty, the user is asked whether to delete this non-empty group. For example, in the editing mode of the Filtering operator shown in Figure 5.5(b), either a form-element or a form-group is associated with a remove button.

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 a move-up button marked “↑↑”

(See Figure 5.5(a)). This operator is used to arrange form-elements/form-groups in the order expected by a transformation. In Figure 5.5(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 a move-down button marked “ \Downarrow ” (See Figure 5.5(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 5.5(a), all elements and groups except the last group **Condition** of the source panel are associated with move-down buttons.

5.3.3 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 Section 5.3.1 and the customization mechanism in Section 5.3.2. Each operator will generate a corresponding formula after a completed customization. We present transformation operators for both value mappings and structural mappings.

Transformation Operators for Value Mappings

Merging Figure 5.4 depicts the interface of a Merging operator. While the source panel contains multiple form-elements for selecting source columns and groups **Delimiter**, which are organized according to the order of the parameters of function *concatenate*, the target panel contains one form-element for selecting a target label. To specify a delimiter, the user ticks one of the pre-defined delimiters (comma, white space, tab, and semicolon) or puts his/her own delimiter in a textbox **Other**.

The mapping formula **Address** = concatenate(C2:C50, “ ”, D2:D50, “ ”, E2:E50) of the running example (Figure 5.3) can be specified as follows. First, cell M2 containing label **Address** and columns C, D, and E are selected; then a 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 5.3.2. Next, two delimiters white spaces “ ” of the two newly added groups are chosen. Finally, button **Submit** is clicked to complete customization.

Splitting Splitting operator is used to express value mappings of the form $s = concatenate(t_1, del_1, \dots, del_{n-1}, t_n)$ where t_i are target atomic labels; s is a source column; and del_j are delimiters between t_j and t_{j+1} , $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, n-1\}$.

Figure 5.6 shows the interface of a Splitting operator. In contrast to Merging operators, 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.

The two mapping formulas 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 cells K2 containing label **FirstName** and L2 containing label **LastName** and activates a 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 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

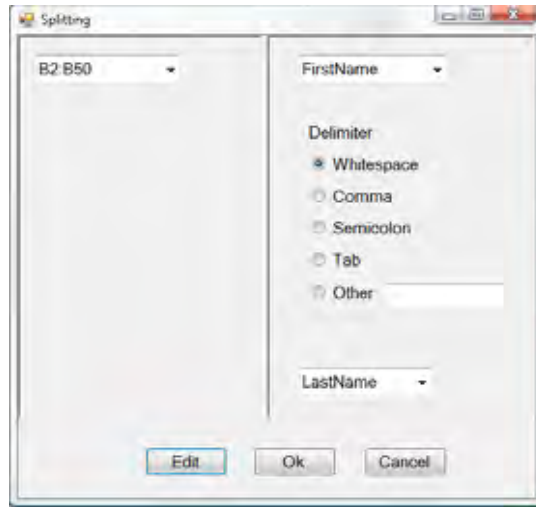


Figure 5.6: Splitting Operator

target element and s is one cell or a collection of cells. The user activates a Copying operator and values of s are copied into values of t . For example, the mapping formula `ProductName =F2:F50` of the running example (Figure 5.1) can be performed by selecting column F and cell N3 and activating a Copying operator.

Transformation Operators for Structural Mappings

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

The interface of a Filtering operator is depicted in Figure 5.5(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 $\{=, <=, >=, \neq, >, <\}$, and a textbox for entering values. The logical operator form-element containing $\{\text{AND}, \text{OR}\}$ is used to combine condition groups.

For example, the mapping formula `Order =A2:H50[AND(D2:D50 = "Sydney", C2:C50 = "Evans")]` of the running example can be specified as follows. The user

| | A | B | C | D | E | F |
|---|----------------|-----------------|-----------------|-----------|------------------|-----------------|
| 1 | <i>OrderID</i> | <i>ProdName</i> | <i>Quantity</i> | <i>ID</i> | <i>FirstName</i> | <i>LastName</i> |
| 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 5.2: Two tables need to be joined

first selects the range A2:H50 and cell J1 and activates a Filtering operator. A form is constructed with one element A2:H50 in the source panel, and the label **Order** in the target panel. Then the user adds a condition group representing $D2:D50 = \text{'Sydney'}$, logical operator element AND, and a condition group representing $C2:C50 = \text{"Evans"}$.

Join. Join operator is used to express structural mappings of the form $t = \text{join}(s_1, s_2, \text{joinexp?})$ 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, a Cartesian product is calculated between s_1 and s_2 . Figure 5.7 illustrates the interface of a Join operator to join two tables in Table 5.2. 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.

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

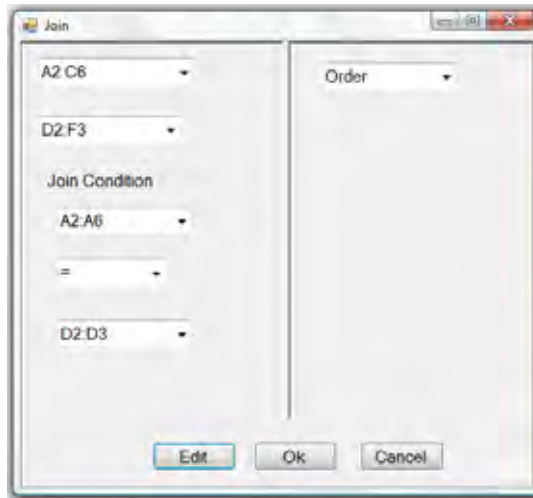
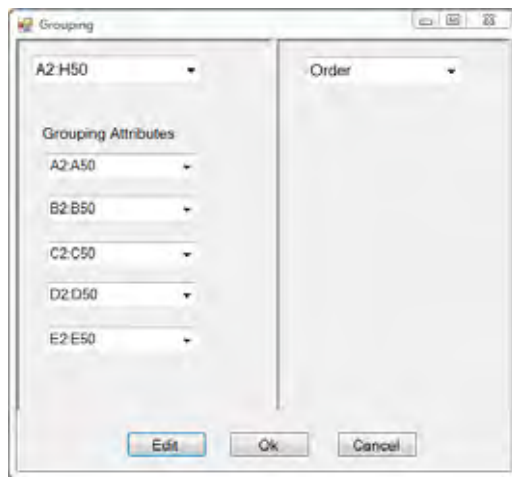
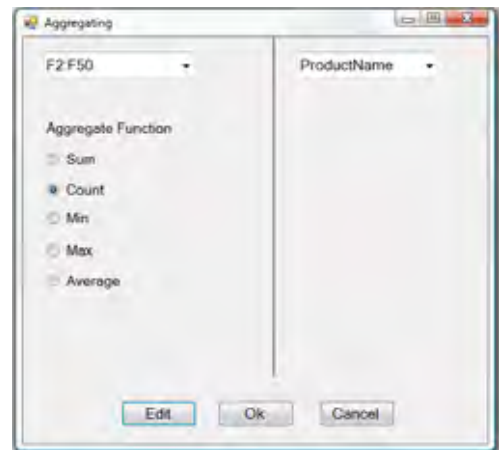


Figure 5.7: Join Operator



(a)



(b)

Figure 5.8: Transformation Operators: (a) Grouping Operator; (b) Aggregate Operator

Figure 5.8(a) illustrates the interface of a 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.

The mapping formula `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 cell J1, and activating a Grouping operator. A form is constructed with the range

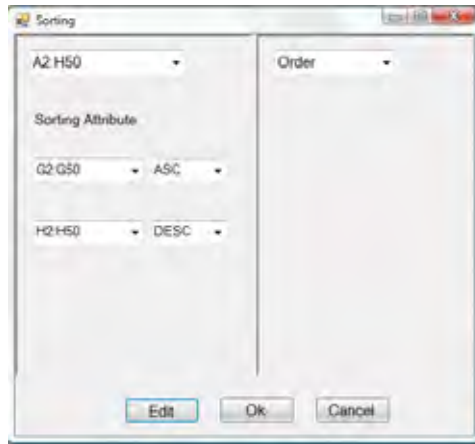


Figure 5.9: Sorting Operator

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.

An 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 $\{sum, count, min, max, average\}$.

Figure 5.8(b) depicts the interface of a Aggregating operator. The mapping formula **ProductName** =count(F2:F50) of the running example can be obtained by selecting row F2:F50 and cell N3 containing label **ProductName**, and activating a Aggregate operator. Then the user needs to insert an aggregate function group, and choose function *count* in the group. The two mapping formulas **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 $o_i \in \{ASC, DESC\}$, $i \in \{1, ..., n\}$.

Figure 5.9 depicts the interface of a 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`.

5.3.4 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 $\{O_1, O_2, \dots, O_n\}$ in sequence (i.e., O_i is performed earlier than O_j if $i < j$), and he/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 [127], specified transformation operations can be seen via connecting lines and functions associated with those lines (See Figure 5.2). 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 he/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 $\text{Id} = \text{A2:A50}$; the last one is filtering with mapping formula $\text{Order} = \text{A2:H50}[\text{AND}(\text{D2:D50} = \text{'Sydney'}, \text{C2:C50} = \text{'Evans'})]$. To modify or delete an operation in the list, the user clicks on the corresponding link under Modify and Delete, respectively.

5.3.5 Expressiveness

The transformation operators presented in Section 5.3.3 are able to cover a set of transformation patterns $\Sigma = \{\text{Copying, Constant Value Generation, Merging, Splitting, Grouping with Aggregation, Join, Catersian Product, Sorting, Filtering}\}$. In fact, a real-world complex transformation scenario is typically composed of multiple transformation patterns. We now describe transformation scenarios, which TranSheet's operators can cover.

Theorem 5.3.1 *TranSheet is able to cover transformation scenarios of the form (S, T, P) where S is a source spreadsheet, T is a target schema, and P is a composition of transformation operations $\{O_1, \dots, O_n\}$:*

- O_i is a transformation operation performed on source columns of S and target labels of T , and O_i belongs to a transformation pattern in Σ , $i \in \{1, \dots, n\}$.
- O_i is performed before O_j if $i < j$.

Proof For each operation O_i ($1 \leq i \leq n$), select source columns and a target label associated with O_i . Then activate a transformation operator that O_i belongs to. Use the operator customization mechanism to customize the newly generated form until obtaining O_i \square .

Note that the expressiveness of TranSheet is not limited by the aforementioned patterns. As more patterns emerge, they can be easily incorporated into TranSheet by designing new operators based on the definition in Section 5.3.1 and the customization mechanism in Section 5.3.2.

As can be observed, so far, we have not dealt with transformation pattern *Derivation*. This pattern is handled in Section 5.4.

5.4 Automated Transformation Suggestions

A Tope is a high-level category of data (e.g., currency, address, phone number, person name) with multiple formats [132] (See Section 2.7 for more details). It also provides functions for automatically transforming among its formats [130]. For example, Tope person name has at least two formats “FirstName LastName” and “LastName, FirstName”. Tope person name also offers two functions for transforming “FirstName LastName” to “LastName, FirstName” and vice versa. In this section, we make use of Topes’ functions to suggest transformations from source columns to atomic target labels. This is helpful when transforming one format to another format using formulas or form-based operators is complicated and difficult.

More specifically, given a Tope T , if a source column c is assigned with format f_1 of T and a target atomic label l_a is associated with format f_2 of T , then transformation from values of c (f_1) to values of l_a (f_2) can be performed using a function of Tope T . For example, with respect to the running example, suppose that two target labels **FirstName** and **LastName** are merged into one label **Name** and this label is associated with format “LastName, FirstName” (e.g., “Prefect, Ford” and “Dent, Arthur”). Meanwhile, the source column B containing person names has format “FirstName LastName” (e.g., “Ford Prefect” and “Arthur Dent”). Transformation from format “FirstName LastName” to format “LastName, FirstName” can be performed via a transformation function of Tope person name.

The main technical problem can be stated as follows. Suppose that target atomic labels are assigned with the formats of a list of Topes and we denote these Topes as set F . Given a source column c , find in F the relevant Topes, whose formats are “closest” to the column c . This can be characterized by function *Recommend*(*ColumnName*, V , F , θ , τ), where:

- *ColumnName* is the name (i.e., header) of the source column c . For example, in the case of the running example, the name of column A (A2:A50) is “OrderID”.
- V is a set of values stored in the column c (e.g., values 42 and 525 of column

Input: *ColumnName*, *V*, *F*, θ , τ

Output: Topes whose formats are closest to the column *c*

```

begin
  foreach T in F do
    if  $ed(T's\ name, ColumnName) > \tau$  then
      |  $F = F - \{T\}$  ;
    end
  end
  foreach T in F do
     $score_T = 0$  ;
    foreach v in V do
      |  $isa(v) = \max\{isa_f(v): f \text{ is a format of } T\}$  ;
      |  $score_T = score_T + isa(v)$  ;
    end
    if  $(score_T/|V|) < \theta$  then
      |  $F = F - \{T\}$  ;
    end
  end
  return F ;
end

```

Algorithm 3: Automated transformation suggestions using Topes.

A).

- *F* is a set of Topes, whose formats are associated with atomic labels of the target schema.
- θ ($\theta \in [0, 1]$) is a threshold to measure similarity between the values of the column *c* and the formats of Topes in *F*.
- τ is an edit distance (i.e., Levenshtein distance) threshold to measure similarity between *ColumnName* and the names of Topes in *F*.

Function *Recommend* is implemented as shown in Algorithm 3. Lines 2-6 of Algorithm 3 compute the edit distance between the name of each tope *T* in *F* and the column name; if the edit distance is greater than τ then *T* is removed from *F*. Lines 7-16 compute the similarity score ($score_T$) between each tope *T* in *F* and values of the column; if the normalized score ($score_T/|V|$) is less than θ , then *T* is removed from *F*. Note that each format *f* of a Tope *T* is associated with a function isa_f [132] that is used to check whether a value *v* matches to *f*: $0 \leq isa_f(v) \leq 1$. If $isa_f(v) = 1$, then *v* totally matches *f*.

In the background, we generate corresponding formulas for transformation functions of topes. For example, the function of Tope person name used to transform format “FirstName LastName” to format “LastName, FirstName” is translated to the following formula: `Name = concatenate(left(B2:B6, search(“ ”, B2:B6)), “, ”, right(B2:B6, len(B2:B6)-search(“ ”, B2:B6)))`.

5.5 Implementation

Figure 5.10 illustrates the architecture of TranSheet for implementing the techniques proposed in this chapter. It extends the architecture presented in Chapter 3 with the following new components:

- **GUI.** A traditional spreadsheet consists of a two-dimensional array of cells, which is unsuitable for displaying nested tables. We extend the tabular grid of MS Excel to display nested tables-based representation of target data.
- **Schema Matching Engine.** This component employs the matching system COMA++ [44] to suggest semantic correspondences between the columns of the source spreadsheet and the labels of the target schema.
- **Transformation Operator Interpretation Engine.** This component interprets form-based transformation operators and generates corresponding mapping formulas.
- **Transformation Recommendation Engine.** This component interacts with the Tope repository and recommends relevant transformations from source columns to target atomic labels. It implements Algorithm 3.
- **Tope Repository.** This repository stores Topes created using the Tope editor [130] or Topes imported from an existing repository available at [27].

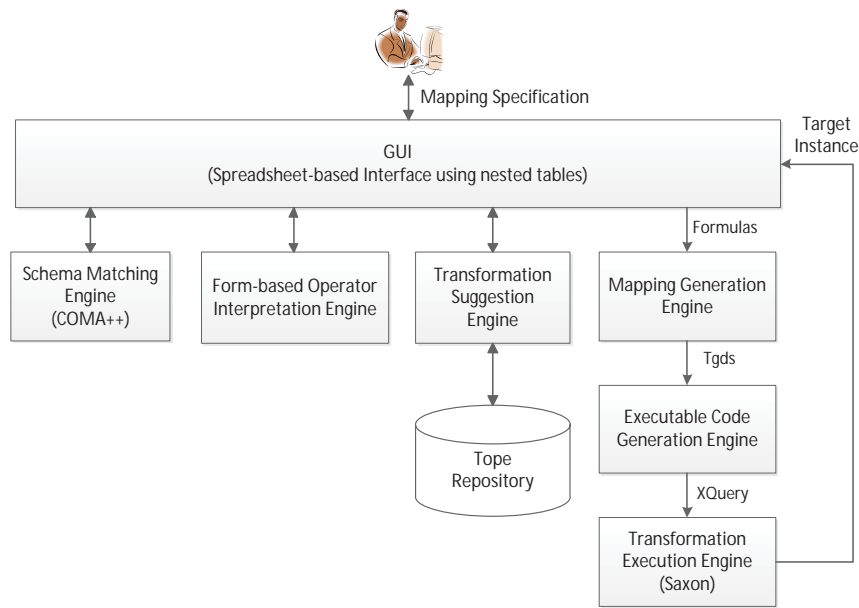


Figure 5.10: Architecture of TranSheet for spreadsheet-based data transformation simplification

5.6 User study

In this section, we focus on conducting a user study to evaluate the usability of TranSheet. We would like to compare TranSheet with a mapping tool. There has been plenty of available mapping tools and they mainly follow the relationship-based paradigm as described in Section 2.5. We chose Altova MapForce 2011 Enterprise Edition (MapForce for short) [1], a popular commercial mapping tool, for comparison since: (i) Unlike research prototypes Clio [79] and Clip [118], MapForce is publically available for download and evaluation; (ii) Unlike commercial mapping tools MS BizTalk Mapper [3] and IBM Rational Data Architect [7], MapForce is easy and quick to install and configure without depending on other complex components. This study is conducted on a laptop with Intel Core Duo 2.1GHz, 3GB RAM, and Windows Vista Home Premium SP2.

5.6.1 Experimental Setup and Methodology

We recruited 10 volunteers without background in using mapping tools, who regularly work with data; each of them has a bachelor degree in various fields, including accountants, programmers, sales, teachers, and project managers. Each participant reported their prior experience with Excel either expert or novice. While novice subjects use spreadsheets to organize information and know some simple functions such as *sum* and *average*, expert subjects can use complex spreadsheet functions on strings, numbers, and conditions (e.g., functions *if*, *and*, *or*...) to manipulate and analyze data. According to these criteria, there are 5 novice subjects and 5 expert subjects.

All subjects had never used MapForce and TranSheet before. A 15-minute tutorial on transformation specification via examples was given to each subject for both TranSheet and MapForce prior to the experiments. Then, subjects demonstrated their understanding by completing a sample transformation themselves with the help of a supervisor if needed.

We then asked subjects to complete eight mapping scenarios (See Table 3.2) used in the expressiveness experiment in Chapter 3. Schemas of the data sets were specified using the Excel-to-XML mapping functionality ¹ of MapForce. We randomized tasks and tools during experimentation for each subject. For each task, we asked subjects to specify mappings and presented to them the final result of the task (i.e., output of a transformation) so that subjects can check the correctness of their specification. We gave enough time to users to understand each task prior to measuring speed. We also provide help manuals to subjects for both tools that could be accessed at any time during each task. Regarding TranSheet, subjects can use formulas, form-based operators, and automated transformation suggestions according to their expertise and preference.

For each task, we recorded the time taken by each subject and charted the mean and median of all subjects for both TranSheet and MapForce. If a subject did not complete a task in 10 minutes, the task was considered incomplete and the time was

¹<http://www.altova.com/mapforce/excel-mapping.html>

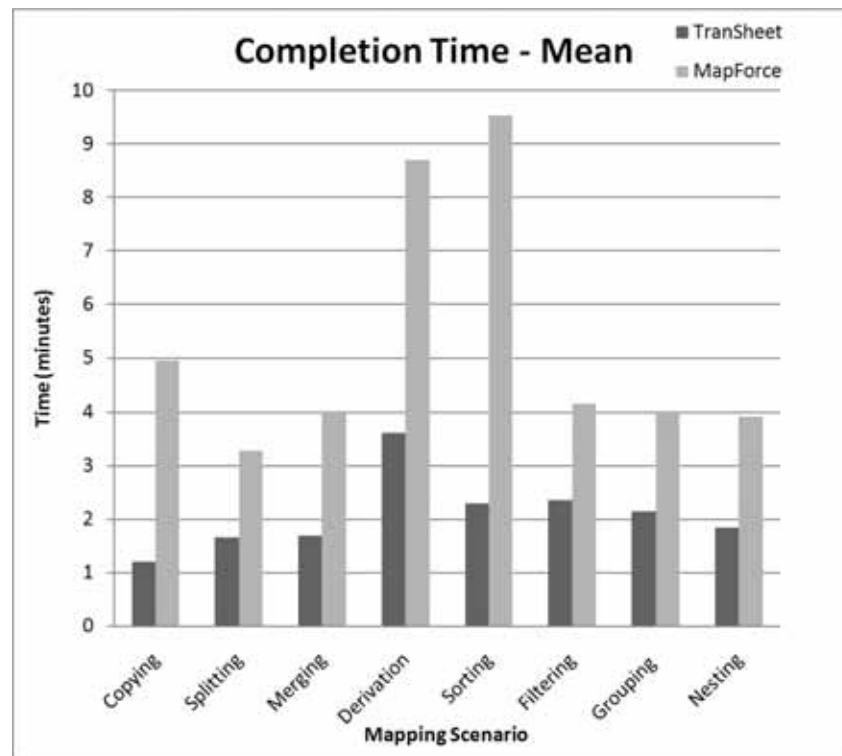


Figure 5.11: User study completion time (mean)

counted as 10 minutes. Finally, each subject completed a post-study questionnaire.

5.6.2 Observations

Figures 5.11 and 5.12 show the mean and median performances of users to complete 8 mapping scenarios in Table 3.2 using TranSheet and MapForce, respectively. In terms of completion time, both mean and median performances of TranSheet were generally over twice as fast as MapForce. By using the Mann-Whitney test, these results are statistically significant with $p\text{-value} < 0.001$ for all mapping scenarios. Based on our observations on users' performance, the main reasons why TranSheet outperforms MapForce can be explained as follows.

Like other *relationship-based* mapping systems [127], the mapping interface of MapForce consists of one schema (the source schema) on the left side and the other schema (the target schema) on the right side. Functions from a predefined library can also be added to the interface; these functions are identical to the ones of the

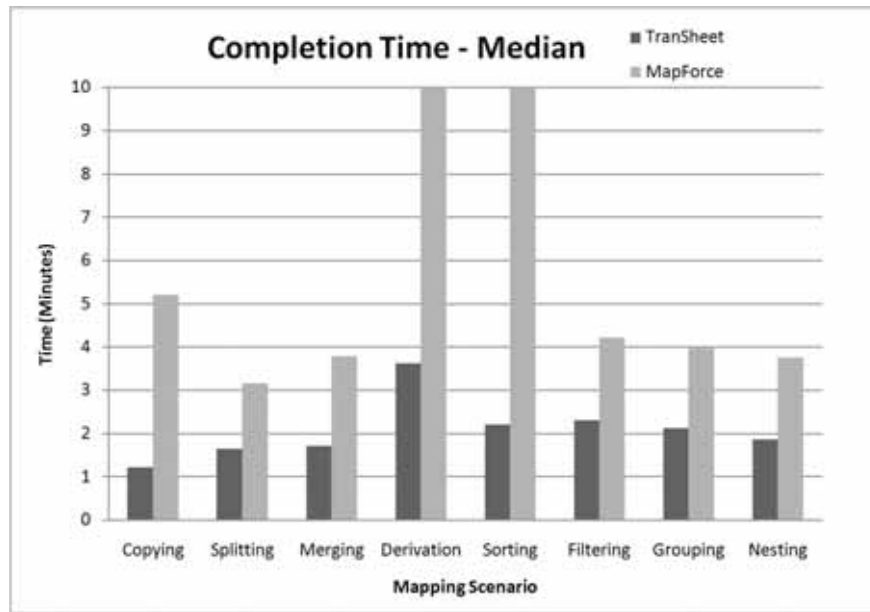


Figure 5.12: User study completion time (median)

programming languages XSLT/XQuery, Java, C#, C++. A function has input and output parameters; each parameter has its own input/output icon. A mapping specification is created by connecting lines between MapForce’s *components* (including the source schema, the target schema, and functions): (i) the source and target schemas; (ii) input(s)/output(s) of a function and the source/target schema; (iii) input(s)/output(s) of two functions.

This leads to some difficulties for spreadsheet users in specifying mappings. First, finding, locating, and using an unfamiliar function as well as connecting lines between components are cumbersome and time-consuming. On the other hand, TranSheet leverages users’ experience with Excel by providing formulas (with many Excel-like functions) and form-based operators. Second, the interface of MapForce is cluttered when multiple functions are used and multiple lines are connected between components (e.g., in the case of mapping scenario derivation). To accomplish a mapping scenario, users often must spend time in reorganizing components and lines in order to understand the details of a mapping specification. Meanwhile, TranSheet avoids this issue since connections between cells and target labels are implicit via formulas, instead of the explicit line connections of MapForce. Third, instant feedback is not provided for each step of a mapping specification in the mapping

interface. Instead, users must click on the Output tab to see the transformation result. In addition to that, source (spreadsheet) data is not available in the Output tab window for users to make side-by-side comparison with target data. Consequently, users waste more time in refining and validating a mapping specification by interacting with three separate windows, namely the mapping interface, Output tab, and the source spreadsheet. By contrast, TranSheet not only retains source data in its mapping interface, but also provides instant feedback for each mapping formula next to source data for comparison convenience. As mappings get more complicated, we expect that the benefits of TranSheet will be more noticeable.

Users completed mapping scenarios copying, derivation, and sorting with TranSheet significantly faster than with MapForce. In the case of copying, while TranSheet supports selecting a subset of source records by using ranges, MapForce requires using three functions, namely *position*, *equal-or-less*, and *filter*.

Only three users could complete scenario derivation with MapForce since five functions must be used (two functions *substring-after*, two functions *substring-before*, and one function *concat*), which makes the mapping diagram complex to understand. Additionally, a function employed twice (e.g., *substring-after* and *substring-before*) in a mapping specification are not renamed by MapForce and cannot be renamed, which also makes users confused. Meanwhile, all users fulfilled this scenario with TranSheet by using automated suggestions; it was difficult to use formulas and form-based operators to implement this scenario.

It is currently impossible to implement mapping scenario sorting using the mapping interface of MapForce so most users could not complete this scenario; only two programmers could complete this scenario by writing XQuery code. In contrast, all users could complete this mapping scenario using TranSheet.

With TranSheet, structural mapping scenarios (e.g., filtering, sorting, grouping with aggregation) basically took more time than value mapping ones (e.g., copying, merging, splitting) since users must perform target schema restructuring to resolve mismatches between spreadsheets and visualization types, in addition to writing formulas or using forms. While Excel experts preferred using formulas, Excel novices frequently relied on form-based wizards.

| Feature | Mean (μ) | Median | Standard Deviation (σ) |
|--|----------------|--------|---------------------------------|
| Formulas | 4.5 | 4.5 | 0.527 |
| Automated suggestions | 4.9 | 5 | 0.316 |
| Form-based operators | 4.6 | 5 | 0.516 |
| Instant feedback with side-by-side comparison | 4.8 | 5 | 0.422 |
| Flowchart-like mapping interface (including lines and functions) of MapForce | 2.1 | 2.5 | 0.944 |

Table 5.3: Users' rating results

5.6.3 Post-study Questionnaire

In addition to completion time evidence, we asked users' opinions on key features that TranSheet and MapForce offer. Each user rated formulas, automated suggestions, form-based operators, instant feedback along with side-by-side comparison, and the flowchart-like mapping interface of MapForce from 1 (not useful) to 5 (most useful).

The rating results are summarized in Table 5.3. As can be seen via mean and median evidence, users rated the features of TranSheet considerably more useful than the flowchart-like mapping interface of MapForce. By performing an ANOVA, there was a significant difference among the ratings ($F_{4,54} = 50.389$, p-value < 0.001). Furthermore, the standard deviations of TranSheet's features are all smaller than the ones of MapForce, demonstrating the consistency of users' opinions on the superiority of TranSheet. These results show the merit of TranSheet's novel features over MapForce particularly and relationship-based mapping tools generally. We also observe that users with a programming background rated the mapping interface of MapForce higher than the other users.

5.7 Related Work

The database, human computer interaction, and machine learning communities have contributed numerous techniques for simplifying data transformation. TranSheet applies a number of these techniques in order to make spreadsheet-based transfor-

mation available to a larger audience.

Form-based interface for querying and code generation

A form-based interface has been successfully used to shield users from the complex syntax of query languages, such as SQL and XQuery [91, 68, 58]. The rationale is that users are good at point-and-click, drag-and-drop, and filling in textboxes to a lesser extent. Thus, this interface is easy-to-use and the learning curve is significantly lower. In addition to that, form-based components are also provided by visual IDEs (e.g., Microsoft Visual Studio [23]) allowing users to connect to database, interact with file systems, and build complex UIs without having to write code from scratch.

In this chapter, we leverage the power of form-based interface to help users graphically specify mappings, instead of writing complex mapping formulas.

Form Customization For Query Specification

The Jayapandian et al. propose an mechanism that enable users to modify an existing form to express the desired query in a flexible manner [92]. The technical sophistication required to modify forms is not much greater than form filling. Inspired by this, we provide a generic mechanism that enables users to customize an existing operator to suit transformation needs.

Transformation Languages and Mapping Tools

Transformation languages, such as XQuery and XSLT, can be used to specify how to transform XML from one format to another. Although such languages are quite powerful (Turing-complete [98]), it can be very difficult, error-prone, and tedious to express a transformation by hand in these languages. To handle this issue, mapping tools (e.g., Clio [85], Clip [117], +Spicy [112], IBM Relational Data Architect [7], Altova MapForce [1], Stylus Studio [9], MS BizTalk Mapper [3]) have been developed to enable users to specify transformations using GUIs. The interface of these mapping tools mainly follows relationship-based metaphor [127]. However, this interface

is typically cluttered and unintuitive for non-technical users.

In this chapter, we proposed some novel techniques built on the top of the spreadsheet-like formula mapping language developed in Chapter 3 in order to make spreadsheet-based data transformation available to non-technical users. We redesigned the mapping interface using nested tables, developed form-based transformation operators, and provided automated transformation suggestions.

Nested tables-based representation

Nested tables-based representation is an intuitive way to display nested relational data. For example, it has been used to develop web applications [142] and mashup programs [139] that are accessible by non-technical users. In this chapter, we use nested tables to represent target data in order to make mapping specification more intuitive.

String-based transformation. Numerous commercial tools and research prototypes provide graphical interfaces allowing users to transform between different string-based formats. Toped++ [130] is an interface for creating Topes [132]; each Tope is a category of data with different formats. A Tope also provides functions for automatically transforming between its formats. Many other transformation tools apply direct manipulation and programming-by-demonstration to specify transformations. Potluck [89] offers simultaneous text editing to transform multiple text fields at the same time. The authors in [134] infer text extractors and transformations for web data from examples of a table. CopyCat [18] applies programming-by-demonstration to data integration via copy and paste actions. Potter’s Wheel [122] provides a collection of form-based operators allowing users to specify transformations graphically. Wrangler [97] extends Potter’s Wheel to provide automated transformation suggestions based on the current context of interaction.

TranSheet leverages some of these techniques to simplify spreadsheet-based transformations. TranSheet uses functions of Topes [132] to automatically suggest transformations. Like Potter’s Wheel [122], TranSheet provides form-based transforma-

tion operators. In addition to that, TranSheet offers a form customization mechanism and uses nested tables to represent target data.

5.8 Summary

In this chapter, we considered the problem of making spreadsheet-based transformation available to non-technical users. We proposed several techniques to simplify mapping specification. We made use of nested tables to reorganize the mapping interface. We integrated a matching module to help users find semantic correspondences between source columns and target labels. We designed form-based transformation operators allowing users to specify transformation graphically, instead of writing complex formulas from scratch. We also provided a form customization mechanism that enables users to customize an existing form to suit transformation needs. Moreover, we relied on Topes to automatically suggest transformations from source columns to atomic target labels.

Our user study indicated that Excel novices and experts can perform data transformation tasks with TranSheet significantly faster than with state-of-the-art mapping tools. The post-study questionnaire also showed that users are more comfortable and satisfied with the novel features of TranSheet in comparison with the mapping interface of mapping tools.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize the main contributions of this dissertation (Section 6.1) and discuss some open future research directions (Section 6.2) to improve this work.

6.1 Concluding Remarks

Spreadsheets are very good at for entering, storing, manipulating, and analyzing small amounts of data. Indeed, a spreadsheet application (e.g., Excel and Open Office Calc) is an ideal database management system (DBMS) [59, 135]. It is a less expensive alternative to larger and more complex DBMSs designed for storing a large amount of data. Spreadsheets also provide a much lower learning curve for non-technical users who wish to study advanced data storage and analysis techniques. Moreover, spreadsheets offer numerous data analysis features which are unavailable in current DBMSs.

Consequently, a significant of the world's data is maintained in spreadsheets by non-technical users. Given the ubiquity and utility of spreadsheets, it has been indispensable for allowing spreadsheet data to interact with external applications and Web services. From Office 2007, Microsoft uses Office Open XML (OpenXML) as the default format for data storage instead of the binary file formats [125]. Recently, OpenXML has been approved by International Organization for Standardization

(ISO) as an international standard. Due to the ubiquity of Excel, this will further facilitate the exchange of spreadsheet data with other applications and Web services.

In this dissertation, we focus on the problem of spreadsheet-based data transformation, which transforms spreadsheet data to the structured formats required by external applications and Web services. We provide a framework with utilities allowing users to specify transformations effectively and easily. Based on investigating the state-of-the-art approaches and conducting informal interviews with data analysts, we have found some main bottlenecks of current solutions to this problem. First, it is challenging for spreadsheet users without programming background to specify transformations and users' spreadsheet programming experience is not leveraged. Second, previously specified transformations are not effectively leveraged to save time and avoid effort duplication. Third, it requires users to modify source spreadsheets to perform transformations, which is tedious and laborious. This may be worsened when a single source spreadsheet must be mapped to multiple target schemas. In order to address these issues, we have proposed a framework, namely TranSheet. This framework consists of necessary components to enable users to: (i) specify transformations in an easy-to-use and convenient manner; (ii) reuse previously specified mappings effectively and efficiently; (iii) avoid cluttering the source spreadsheet with transformations by embedding transformation logic into an expressive language. We mainly build our framework on top of Excel, due to its ubiquity; but the techniques and concepts presented in the dissertation are also applicable to other spreadsheet applications, such as Open Office Calc [25] and Apple Numbers [20]. In the following, we summarize the most significant contributions of this dissertation.

6.1.1 An expressive and familiar spreadsheet-like formula mapping language

In order to enable users to transform spreadsheet data to structured formats, we develop a familiar and expressive spreadsheet-like formula mapping language (Chapter 3). In particular, we make the following contributions:

- A *spreadsheet-like formula mapping language* has been designed for specifying mappings between spreadsheet data and the target schema. In terms of *expressiveness*, we have demonstrated that popular transformation patterns that are relevant to spreadsheet-based transformation are supported in the language using spreadsheet formulas and functions. These patterns are the result of a careful analysis of commonly needed mapping scenarios supported by transformation languages, mapping tools, and spreadsheet corpuses. This enables the language to avoid cluttering the source spreadsheet with transformations and it turns out to be helpful when multiple target schemas are mapped (e.g., visualize a data set using multiple visualization types).
- *Target schema restructuring* has been proposed to allow users to resolve the mismatches between the source spreadsheet and the target schema. It is a common occurrence that the target schema, which is defined externally, does 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 how mapping formulas should be interpreted.
- *Mapping generalization constructs* have been proposed to enable users to generalize a mapping from instance-level to template-level element. This allows the mapping to be applied to multiple spreadsheet instances with similar structure. Frequently used formatting features of spreadsheet templates are exploited to generalize mappings. More specifically, the mapping generalization mechanism allows specifying relative locations of spreadsheet data, expressing dynamic length ranges, and mapping of non-adjacent collections of cells.
- *Tuple generating dependencies (tgds)*, a widely used schema mapping formalism, have been used to describe the semantics of the language. In comparison with the state-of-the-art schema mapping and data exchange, we have introduced a collection of new functions to tgd expressions. We then have extended a previous query generation algorithm to generate executable script for these new functions. Each generated target document of TranSheet corresponds to a canonical universal solution of data exchange.

- A prototype of the language has been implemented as an Excel plug-in. We have evaluated the expressiveness and mapping generalization of TranSheet in two real applications, namely end-user visualization and medical data transfer. The experimental results show that our language is expressive and flexible enough to support numerous practical spreadsheet-based transformation scenarios.

6.1.2 Previously specified mapping formulas reuse recommendation

In order to enable users to reuse previously specified mapping formulas of the language presented in Chapter 3, we have proposed a solution (Chapter 4), which consists of the following contributions:

- We have formulated the problem of spreadsheet-based transformation reuse as a variant of *similarity join*, which is a well-known similarity search problem that find all pairs of objects whose similarity is above a given threshold. To the best of our knowledge, this problem has not been characterized before in the setting we consider here.
- We have defined spreadsheet templates that are used to characterize spreadsheet structures. In many cases, spreadsheets evolve in a number of predictable ways and various spreadsheets tend to emerge from a common pattern. We have also proposed novel techniques to infer a template from an existing spreadsheet based on common spreadsheet presentation patterns. Then, we have provided an algorithm to generate the string-based representation of an inferred template.
- We have proposed an algorithm to recommend previously specified mappings for a new spreadsheet instance that needs to be mapped to the target schema. This relies on computing similarities between string-based representations of templates of the new instance and previously specified instances.

- A *mapping repository* has been designed to store schema, spreadsheet, template, and mapping information. We have implemented a prototype for the proposed techniques by extending the architecture described in Chapter 3. We then have evaluated the performance of our solution using spreadsheets from a real corpus. The experimental results show that our solution is efficient and effective enough to support a few hundred thousand mappings stored in the mapping repository.

6.1.3 End-user oriented spreadsheet-based transformation techniques

In order to make spreadsheet-based data transformation available to non-technical users, we have proposed a number of novel techniques (Chapter 5), which extend the formula mapping language presented in Chapter 3. More specifically, we make the following contributions:

- The interface has been redesigned based on *nested tables* to make it more intuitive and convenient for users to specify and refine transformations. Nested tables have been successfully used to help non-technical users develop web and mashup applications. With this new interface, users can not only preview the whole transformation result, but also make side-by-side comparison between the source and target data. A matching module has been integrated to allow users to semi-automatically find semantic correspondences between the source spreadsheet columns and the target schema labels. As described in Chapter 2, matching is a time-consuming and labour-intensive task.
- A set of *form-based transformation operators* have been designed allowing users to specify transformations graphically, instead of writing complex mapping formulas from scratch. The benefits of these operators are two-fold: (i) users with no deep spreadsheet programming background can specify transformations without having to remember complex syntax of the language; (ii) spreadsheet programming experts can boost their productivity. We have also

provided a form customization mechanism allowing users to customize an existing transformation operator to suit transformation needs. This is needed because when an operator is activated from a contextual menu, users often require modifying this operator. Unlike relationship-based mapping interface, users cannot see specified transformation operations in the mapping interface of TranSheet. As a result, we have offered a history list allowing users to modify these operations.

- Sometimes, specifying transformations using formulas and form-based operators (e.g., transforming format “firstname lastname” to format “lastname, firstname”) may be difficult and complicated. Therefore, we have provided a *automated transformation suggestions* mechanism by making use of Topes; each Tope is a category of data with a number of formats and functions for transforming among these formats.
- We have implemented a prototype for the above techniques and conducted an extensive user study to evaluate the usability of TranSheet. The experimental results show that TranSheet significantly reduces specification time and promotes users’ satisfaction in transformation specification in comparison with state-of-the-art mapping tools.

6.2 Future Directions

In this dissertation, we have studied the problem of spreadsheet-based data transformation. As we have shown in the whole dissertation, this is an important and real research problem. TranSheet can be extended and improved in numerous ways. In this section, we discuss some possible future extensions to our work.

Extracting structured data from spreadsheets. In the whole dissertation, we assume that the schemas of spreadsheets are available. Spreadsheet applications offer users a high level of flexibility in terms of data formatting. This flexibility allows users to organize data according to their own preferences, styles, and subjective importance. However, the downside is that this freedom raises an ample challenge

in identifying spreadsheet schemas since spreadsheets are typically unstructured. However, the state-of-the-art provides a limited support to address this challenge. This will be an important, but challenging research direction.

Applying machine learning techniques to simplify data transformation.

Given a set of input examples, machine learning techniques recognize complex patterns and make intelligent decisions based on data. These techniques can be used to simplify transformation specification in several ways.

First, instead of specifying transformations using formulas or form-based wizards, users can create a new transformation rule by providing sample source data and sample target data. A learning algorithm will infer a relevant transformation rule from these samples. Users can then apply this inferred rule formulas to current transformation tasks. For example, a user can provide two sample source values “Bill Gates” and “Steve Jobs” and two sample target values “Bill” and “Steve”, a program can generate a transformation rule with mapping formulas to perform splitting.

Second, it may be hard for non-technical users to specify a generic mapping formula (e.g., `ProdName = A1:A<next=bottom(Orders) + 2>`) using the mapping generalization constructs as presented in Section 3.5. A possible solution is users provide a collection of sample spreadsheets belonging to a same template and a program automatically infers an appropriate generic formula by learning the structure of the template via these sample templates.

A formal description of TranSheet’s expressiveness. It is known that current tgd-based mapping systems (Clio, Clip, +Spicy, and TranSheet) are not Turing-complete [143]. As a result, they are not as expressive as XSLT and XQuery, which are Turing-complete languages. More importantly, a formal and exact description of mapping scenarios that can be handled by these tgd-based mapping systems is still missing. This is an open problem for schema mapping systems including TranSheet. For example, STBenchmark [37], a benchmark for mapping systems, just proposes transformation scenarios based on a careful analysis of elementary constructs needed applications such as data exchange, data warehouses, XML publishing, schema evolution, and real-world mapping specifications.

This will help theoretically answer the question whether transformations (a single transformation or a combination of transformations) of TranSheet preserve the semantics of the original data source. Currently, proposed transformations are based on practicality rather than theoretical backgrounds.

A benchmark for spreadsheet-based data transformation. Based on our investigation, we believe it is crucial to develop a set of standard mapping scenarios that is relevant to spreadsheet-based data transformation. Such standard will serve to standardize the specification of basic mapping scenarios for spreadsheet-based data transformation. For example, STBenchmark [37] is the first effort towards the development of a uniform testbed and repository for schema mapping and data exchange tasks.

Supporting transforming spreadsheet to different formats. In addition to XML, TranSheet can be extended to support transforming spreadsheet data to different formats: (i) relational data stored by MS SQL Server, IBM DB2, Oracle, and MySQL; (ii) Flat (text) files; (iii) Electronic Data Interchange (EDI) standards (e.g., UN/EDIFACT, ANSI X12, Health Level 7, SAP IDoc, and IATA PADIS); (iv) Extensible Business Reporting Language (XBRL); (v) JavaScript Object Notation (JSON) [29].

Transforming external structured data to spreadsheet data. On the reverse side of transforming spreadsheet data to external structured formats, TranSheet can be extended to transform external structured data (e.g., flat files, XML, and relational data) to spreadsheet data. For example, Altova MapForce [1] supports both sides of transformation (i.e., spreadsheet data can be used as either mapping sources or mapping targets).

Publications

1. Regis Saint-Paul, Vu Hung, G. Al-Naymat, and Boualem Benatallah. Spreadsheet-based complex data transformation. Technical report, CSE-UNSW-0919-2009.
2. Vu Hung, Boualem Benatallah, and Regis-Saint Paul. Spreadsheet-based complex data transformation. CIKM 2011, Glasgow, UK, 24-28 October, 2011.
3. Vu Hung and Boualem Benatallah. Spreadsheet-based data transformation. Information System Journal, 2011 (submitted)
4. Vu Hung and Boualem Benatallah. Spreadsheet-based data transformation reuse. ICSE (International Conference on Software Engineering) 2012, Zurich, Switzerland, June 2 - 9 (To be submitted).

Appendix A

Appendix

A.1 Template inference based on cp-similarity

Abraham et al [35] presents an inference technique by identifying similarity regions. Two formulas are *cp-similar* if their R1C1-style presentations are the same [54]. In other words, their formulas may have resulted from a copy/paste action from one of the cells to the other. Recall that spreadsheets have two popular notations for representing formulas, namely A1 and R1C1. Although the A1 notation is dominating, the meaning of a formula representing by the R1C1 notation is independent of the cell which stores it. In the R1C1 notation, both rows and columns are numbered by integers from 1 onward. For arbitrary nonzero integers i and j and positive integers m and n , the following expressions are valid: RmCn, R[i]Cn, RmC[j], R[i]C[j], RCn, RC[j], RmC, R[i]C. While the number after ‘R’ refers to the row number, the number after ‘C’ refers to the column number. If that number is absent, it means same row (column) as the cell in which this expression is used. A number in the square brackets is a relative reference and the cell to which this expression points should be identified by adding that number to the row (column) of the present cell. Number without brackets is an absolute reference to a cell whose row (column) number is equal to that number [135]. For example, R[-1]C1 denotes a cell which is in the row directly above the present one in column 1.

The cp-similar cells are then grouped on the basis of rows and columns to identify

| | A | B | C | D | E | F | G |
|---|-------|----------|------|-------|----------|------|-------|
| 1 | | 2005 | | | 2006 | | |
| 2 | Name | Quantity | Cost | Total | Quantity | Cost | Total |
| 3 | Towel | 3 | 4 | 12 | 2 | 1 | 2 |
| 4 | Soap | 2 | 5 | 10 | 3 | 2 | 6 |
| 5 | Fish | 1 | 6 | 6 | 3 | 6 | 18 |
| 6 | | | | | | | |

(a)

```

Target
Products[0..*]
  Year = B1:(next=right(Year)+3)1
Items[0..*]
  Name = A3:A(value=empty)
  Quantity = (right(Year))3:right(Year)(value=empty)
  Cost = (right(Year)+1)3:(right(Year)+1)(value=empty)
  Total = (right(Year)+2)3:(right(Year)+2)(value=empty)

```

(b)

Figure A.1: (a) Sales data; (b) Mapped target schema

| | A | B | C | D |
|---|-------|----------|------|--------|
| 1 | | 2005 | | |
| 2 | Name | Quantity | Cost | Total |
| 3 | Towel | 3 | 4 | =B3*C3 |

Figure A.2: Automatically Inferred Template

hex groups and vex groups. Then, the system tries to overlay identified rows and columns to generate the template. For each pair of rows/columns that is considered for overlay, in addition to the formula cells, referenced data cells in two rows/columns are checked to find out if they have the same type. While vex groups are inferred by overlaying blocks of cells vertically, hex groups are inferred by overlaying blocks of cells horizontally.

Let the predicate c test for cp-similarity and t test for type similarity. $c(f, f')$ means that f and f' are cp-similar; $t(v, v')$ implies that v and v' have the same type. A spreadsheet is given by a function $S : A \rightarrow F$ mapping cell addresses to values and formulas. A cell of the spreadsheet is given by $(a, f) \in S$ where $S(a)$ provides yields formulas/values at address a . Let d_r be a function that $d_r(a_1, a_2)$ gives the row difference between the cell addresses a_1 and a_2 ; let d_c be a function such that $d_c(a_1, a_2)$ yields the column differences between the cell addresses a_1 and a_2 .

Template of spreadsheet in Figure A.1(a) can be inferred as follows. Hex groups are identified based on: $c(S(D_3), S(G_3)) \wedge c(S(D_4), S(G_4)) \wedge c(S(D_5), S(G_5))$; $d_c(D_3, G_3) = d_c(D_4, G_4) = d_c(D_5, G_5) = 3$; $t(B_3, E_3) \wedge t(B_4, E_4) \wedge t(B_5, E_5)$; $t(C_3, F_3) \wedge$

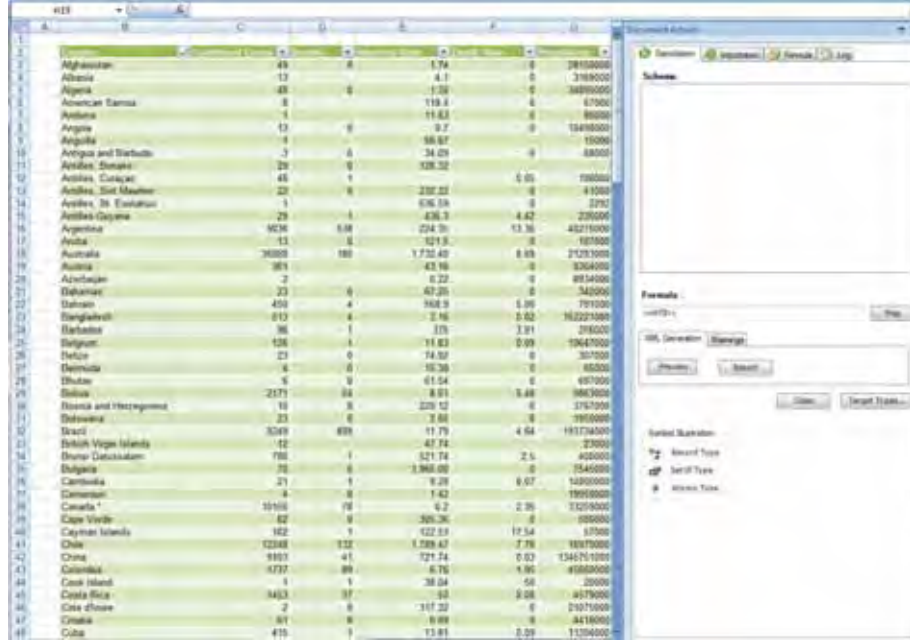


Figure A.3: Interface for mapping swine flu data to a pie chart

$t(C_4, F_4) \wedge t(C_5, F_5)$. Remember that in terms of cp-similarity, we have: $S(D_3) = S(D_4) = S(D_5) = S(G_3) = S(G_4) = S(G_5) = RC[-2] * RC[-1]$. Once the overlay for the hex group has been done, vex groups are identified based on: $c(S(D_3), S(D_4)) \wedge c(S(D_3), S(D_5))$; $t(B_3, B_4) \wedge t(B_3, B_5)$; $t(C_3, C_4) \wedge t(C_3, C_5)$. The inferred template for the instance in Figure A.1(a) is shown in Figure A.2. The system shades vex groups green and hex groups light pink, respectively; template cells that are part of vex and hex groups are shaded blue respectively.

A.2 TranSheet's screenshots

In this section, we demonstrate step-by-step with screenshots the functionalities of TranSheet via two mapping scenarios.

Mapping swine flu data to a pie chart

While spreadsheet data is on the left side, the target schema is located in the Excel task pane on the right side (See Figure A.3). Spreadsheet data can be imported using the built-in functionality of Excel.

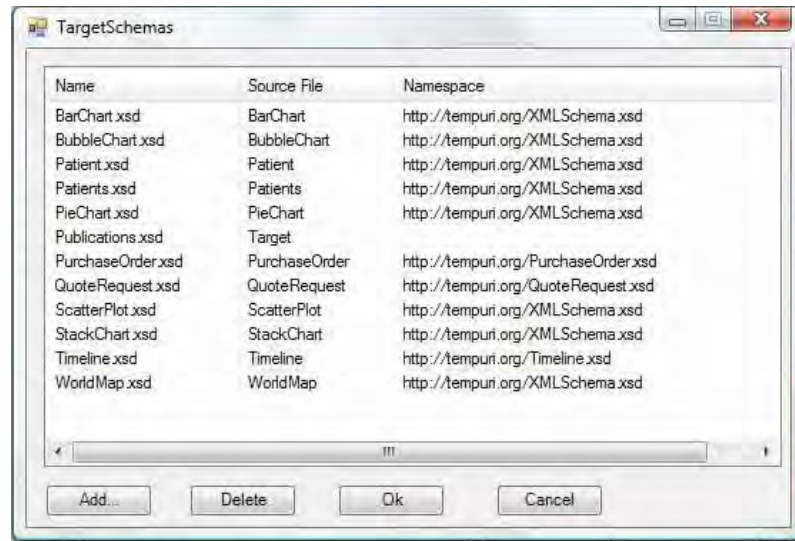


Figure A.4: Schema repository of TranSheet

Target schemas are imported using TranSheet functionality by clicking on button **Target Types** as shown in Figure A.4. The user can select an existing schema from the repository or add a new schema by clicking on button **Add**.

In this example, the target schema of visualization type pie chart is selected and visualized as a tree in the task pane (See Figure A.5). The user can start specifying a mapping by selecting a target label and entering a formula into the formula editor.

In this example, the user wants to use a pie chart to visualize confirmed cases for 193 countries. There are two ways of specifying formulas, using either a structural mapping or two value mappings:

- **Pies** =B3:C195
- **Name** =B3:B195; **Value** =C3:C195

Instant feedback is displayed next to the corresponding labels of the target schema as shown in Figure A.6. The user can validate and refine the specified mapping based on this feedback.

After completing mapping, the user can preview the target document before exportation by clicking on button **Preview** as shown in Figure A.7. Now the generated XML document can be exported for use with other applications.

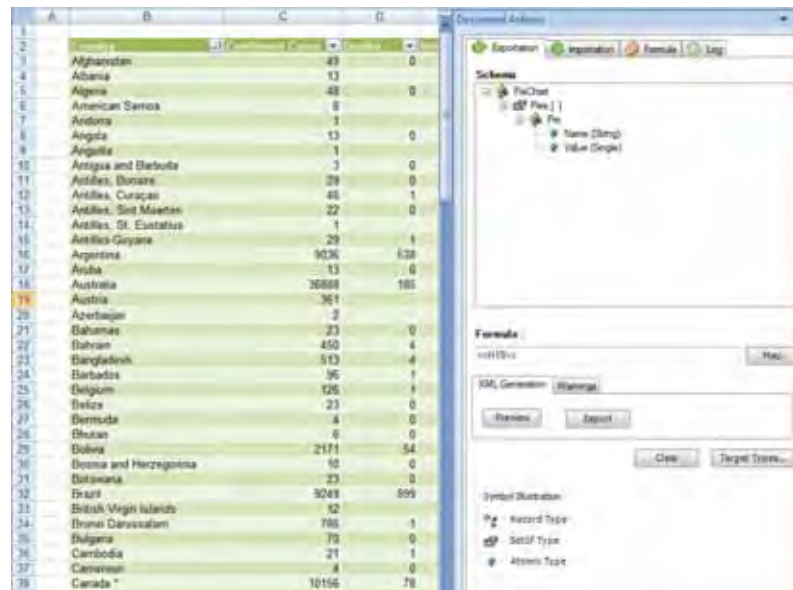


Figure A.5: Interface for mapping swine flu data to a pie chart with the selected target schema

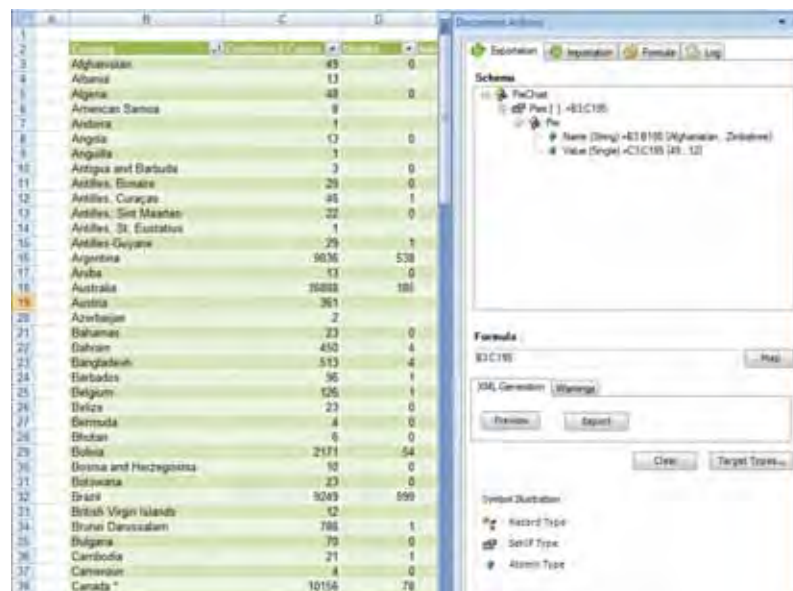


Figure A.6: Interface for mapping swine flu data to a pie chart with instant feedback for mappings

```
<?xml version="1.0" encoding="utf-8" ?>
<?PacChart>
  <Pac>
    <Name>Afghanistan</Name>
    <Value>49</Value>
  </Pac>
  <Pac>
    <Name>Albania</Name>
    <Value>13</Value>
  </Pac>
  <Pac>
    <Name>Algeria</Name>
    <Value>40</Value>
  </Pac>
  <Pac>
    <Name>American Samoa</Name>
    <Value>0</Value>
  </Pac>
  <Pac>
    <Name>Andorra</Name>
    <Value>1</Value>
  </Pac>
  <Pac>
    <Name>Angola</Name>
    <Value>13</Value>
  </Pac>
  <Pac>
    <Name>Anguilla</Name>
    <Value>1</Value>
  </Pac>
  <Pac>
    <Name>Antigua and Barbuda</Name>
    <Value>3</Value>
  </Pac>

```

Figure A.7: Transformation preview of TranSheet

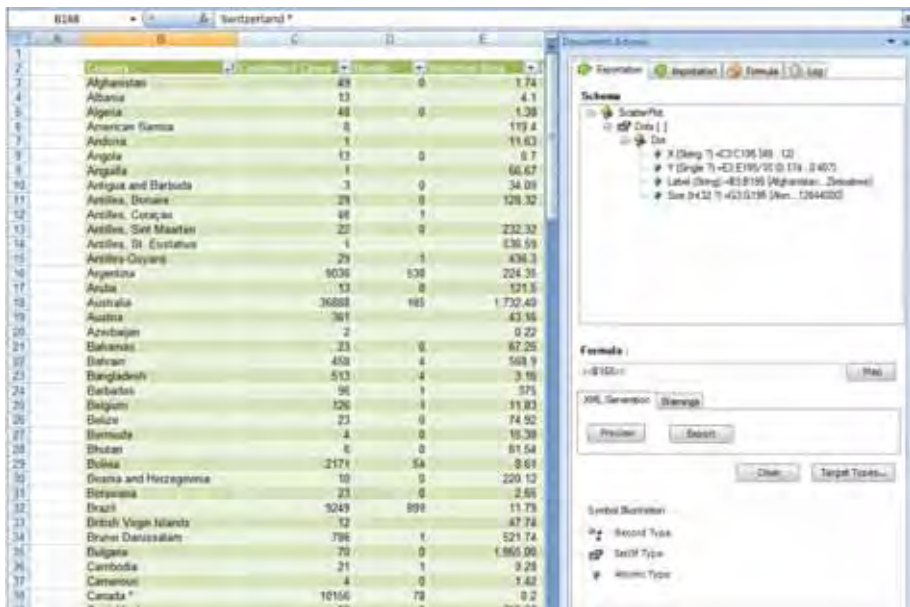


Figure A.8: Interface for mapping swine flu data to a scatter plot

Mapping swine flu data to a scatter plot

In this scenario, the user wants to use a scatter plot to find correlation between the number of confirmed cases and the infection rate in each country (See Figure A.8). The infection rate is visualized per one hundred thousand, instead of per million in the data set. The following mappings are specified:

- `X = C3:C195`
- `Y = E3:E195/10`
- `Label = B3:B195`
- `Size = G3:G195`

Bibliography

- [1] *Altova MapForce Professional Edition, 2011 Version*. <http://www.altova.com>.
- [2] Atova mapforce. <http://www.altova.com/mapforce.html>.
- [3] *Creating Maps Using BizTalk Mapper, BizTalk 2004*.
<http://msdn.microsoft.com/en-us/library/ms943073.aspx>.
- [4] Excel web app. <http://office.live.com>.
- [5] Google fusion tables. <http://tables.googlelabs.com>.
- [6] Google spreadsheet. <http://docs.google.com>.
- [7] *IBM Rational Data Architect*. <http://www-01.ibm.com/software/data/integration/rda/>.
- [8] Manyeyes. <http://many-eyes.com/>.
- [9] *Stylus Studio, XML Enterprise Suite, Release 2*.
<http://www.stylusstudio.com>.
- [10] Special issue on data transformation. *IEEE Data Eng. Bull.*, 22(1), 1999.
- [11] *XML Path Language Version 1.0*. <http://www.w3.org/TR/xpath>, November 1999.
- [12] *XSL Transformations (XSLT) Version 1.0*. <http://www.w3.org/TR/xslt>, 1999.

- [13] Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.
- [14] Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 2007.
- [15] *XML Path Language Version 2.0*. <http://www.w3.org/TR/xpath20/>, January 2007.
- [16] *XQuery 1.0*. <http://www.w3.org/TR/xquery/>, January 2007.
- [17] *XSL Transformations (XSLT) Version 2.0*. <http://www.w3.org/TR/xslt20/>, January 2007.
- [18] *Interactive data integration through smart copy and paste*. www.crdldb.org, 2009.
- [19] Open information integration. <http://openintegration.org/>, December 2009.
- [20] Apple numbers. <http://www.apple.com/iwork/numbers/>, July 2010.
- [21] Freebase. <http://www.freebase.com/>, July 2010.
- [22] Girdworks. <http://code.google.com/p/freebase-gridworks/>, July 2010.
- [23] Ms visual studio. <http://www.microsoft.com/visualstudio>, August 2010.
- [24] Open information integration project. <http://openintegration.org/>, June 2010.
- [25] Openoffice. <http://www.openoffice.org/>, July 2010.
- [26] Saxon engine. <http://saxon.sourceforge.net/>, June 2010.
- [27] Tope software. <http://www.topedepot.info/>, September 2010.
- [28] Gnumeric. <http://projects.gnome.org/gnumeric/>, March 2011.
- [29] Json format. <http://www.json.org/>, January 2011.

- [30] Open information integration code. <http://openii.sourceforge.net/>, February 2011.
- [31] Xsl transformations (xslt) version 2.0. <http://www.w3.org/TR/xslt20/>, January 2011.
- [32] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [33] Serge Abiteboul and Nicole Bidoit. Non first normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci.*, 33(3):361–393, 1986.
- [34] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 165–172, Washington, DC, USA, 2004. IEEE Computer Society.
- [35] Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 182–191, New York, NY, USA, 2006. ACM.
- [36] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The claremont report on database research. *SIGMOD Rec.*, 37(3):9–19, 2008.
- [37] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *VLDB'08*.
- [38] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. Comparing and evaluating mapping systems with stbenchmark. *Proc. VLDB Endow.*, 1(2):1468–1471, 2008.

- [39] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, November 2003.
- [40] Altov. Altovaxml. <http://www.altova.com/altovaxml.html>, February 2011.
- [41] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 40–49, Washington, DC, USA, 2008. IEEE Computer Society.
- [42] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *Proc. VLDB Endow.*, 2(1):514–525, 2009.
- [43] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB '06*.
- [44] David Aumüller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 906–908, New York, NY, USA, 2005. ACM.
- [45] Andrey Balmin, Latha Colby, and Emiran Curtmola. Seda: A system for search, exploration, discovery, and analysis of xml data. 2008.
- [46] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.
- [47] P. A. Bernstein. Generic model management: A database infrastructure for schema manipulation, September 2003. Keynote Address, IDM 2003 Workshop, Seattle, Washington, September 2003.
- [48] P.A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proc. of the 2007 ACM SIGMOD Int'l. Conf. on Management of Data*, pages 1–12, 2007.
- [49] Philip A. Bernstein and Laura M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, 2008.

- [50] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, and Christoph Quix. Industrial-strength schema matching. *SIGMOD Rec.*, 33(4):38–43, 2004.
- [51] Angela Bonifati, Giansalvatore Mecca, Alessandro Pappalardo, Salvatore Raulich, and Gianvito Summa. The spicy system: towards a notion of mapping quality. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1289–1294, New York, NY, USA, 2008. ACM.
- [52] Bob Brauer. Next evolution of data integration into microsoft excel. Technical report, StrikeIron, 2005.
- [53] Bob Brauer. Next evolution of data integration into microsoft excel. Technical report, StrikeIron Inc., 2006.
- [54] Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel. Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. *IEEE Trans. Softw. Eng.* 2006.
- [55] Michael Carey. Data delivery in a service-oriented world: the bea aqualogic data services platform. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 695–705, New York, NY, USA, 2006. ACM.
- [56] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An xml query language for heterogeneous data sources. pages 53–62. Springer-Verlag, 2000.
- [57] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE '06*.
- [58] Joobin Choobineh, Michael V. Mannino, and Veronica P. Tseng. A form-based approach for database analysis and design. *Commun. ACM*, 35:108–120, February 1992.
- [59] Paul Cornell. *Excel as Your Database*. Appress, 2007.
- [60] Altova Corp. Excel mapping. <http://www.altova.com/mapforce/excel-mapping.html>, January 2011.

- [61] Microsoft Corporation. Ms .net framework. <http://msdn.microsoft.com/en-us/netframework>, March 2011.
- [62] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.
- [63] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for xml. In *WWW*, 2008.
- [64] Hong-Hai Do and Erhard Rahm. Coma: a system for flexible combination of schema matching approaches. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 610–621. VLDB Endowment, 2002.
- [65] AnHai Doan. *Learning to Map between Structured Representations of Data*. PhD thesis, University of Washington, 2002.
- [66] Andrew Eisenberg and Jim Melton. Sql/xml is making good progress. *SIGMOD Rec.*, 31(2):101–108, 2002.
- [67] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19:1–16, 2007.
- [68] David W. Embley. Nfql: the natural forms query language. *ACM Trans. Database Syst.*, 14:168–211, June 1989.
- [69] Martin Erwig, Robin Abraham, Irene Cooperstein, and Steve Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 136–145, New York, NY, USA, 2005. ACM.
- [70] Ronald Fagin. Inverting schema mappings. *ACM Trans. Database Syst.*, 32(4):25, 2007.

- [71] Ronald Fagin, Phokion G. Kolaitis, Rene J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005.
- [72] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 2005.
- [73] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.
- [74] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.
- [75] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Quasi-inverses of schema mappings. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 123–132, New York, NY, USA, 2007. ACM.
- [76] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, New York, NY, USA, 2006. ACM Press.
- [77] Marc Fisher and Gregg Rothermel. The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes'05*.
- [78] Apache Software Foundation. Xalan apache. <http://xalan.apache.org/>, March 2011.
- [79] Ariel Fuxman, Mauricio A. Hernandez, Howard Ho, Renee J. Miller, Paolo Papotti, and Lucian Popa. Nested mappings: schema mapping reloaded. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 67–78. VLDB Endowment, 2006.

- [80] Avigdor Gal. The generation y of xml schema matching panel description. In *XSym*, pages 137–139, 2007.
- [81] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB '01*.
- [82] T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
- [83] Marc Gyssens, Laks V. S. Lakshmanan, and Iyer N. Subramanian. Tables as a paradigm for querying and restructuring (extended abstract). pages 93–103, 1996.
- [84] L. M. Haas, R. J. Miller, B. Niswonger, M. Tork, Roth P. M. Schwarz, and E. L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22:31–36, 1999.
- [85] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 805–810, New York, NY, USA, 2005. ACM.
- [86] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 9–16. VLDB Endowment, 2006.
- [87] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise information integration: successes, challenges and controversies. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 778–787, New York, NY, USA, 2005. ACM.
- [88] Mauricio A. Hernandez, Paolo Papotti, and Wang-Chiew Tan. Data exchange with data-metadata translations. 2008.

- [89] David Huynh, Robert Miller, and David Karger. Potluck: Data mash-up for non-programmers. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon J B Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Guus Schreiber, and Philippe Cudr-Mauroux, editors, *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007)*, Busan, South Korea, volume 4825 of *LNCIS*, pages 239–252, Berlin, Heidelberg, November 2007. Springer Verlag.
- [90] Google Inc. Google refine. <http://code.google.com/p/google-refine/>, March 2011.
- [91] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24, New York, NY, USA, 2007. ACM.
- [92] Magesh Jayapandian and H. V. Jagadish. Expressive query specification through form customization. *EDBT*, 2008.
- [93] 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*.
- [94] Haifeng Jiang, Howard Ho, Lucian Popa, and Wook-Shin Han. Mapping-driven xml transformation. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1063–1072, New York, NY, USA, 2007. ACM.
- [95] S. Jones, A. Blackwell, and M. Burnett. A user-centered approach to functions in excel. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 165–176. ACM Press, 2003.
- [96] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centered approach to functions in excel. In *ICFP'03*.

- [97] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011.
- [98] Stephan Kepser. A proof of the turing-completeness of xslt and xquery. In *Technical report SFB 441, Eberhard Karls Universitat Tübingen*, 2004.
- [99] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning*. Wiley, 2004.
- [100] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 61–75, New York, NY, USA, 2005. ACM.
- [101] Woralak Kongdenfha, Boualem Benatallah, Julien Vayssier, Régis SaintPaul, and Fabio Casati. Rapid development of spreadsheetbased web mashups. *WWW*, 2009.
- [102] Woralak Kongdenfha, Boualem Benatallah, Julien Vayssière, Régis Saint-Paul, and Fabio Casati. Rapid development of spreadsheet-based web mashups. In *WWW*, 2009.
- [103] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.*, 3:484–493, September 2010.
- [104] L.V.S. Lakshmanan, S.N. Subramanian, N. Goyal, and R. Krishnamurthy. On querying spreadsheets. In *ICDE'98*.
- [105] Maurizio Lenzerini. Data integration: a theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM.
- [106] Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE '09*.

- [107] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Halevy. Corpus-based schema matching. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 57–68, Washington, DC, USA, 2005. IEEE Computer Society.
- [108] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *The VLDB Journal*, pages 49–58, 2001.
- [109] Jayant Madhavan, Shawn R. Jeffery, Shirley Cohen, Xin (luna) Dong, David Ko, Cong Yu, Alon Halevy, and Google Inc. Web-scale data integration: You can only afford to pay as you go. In *Proc. of CIDR-07*, 2007.
- [110] Benoit Marchal. Working xml: Comparing xslt 2.0 and xquery. *developer-Works*, April 2006.
- [111] Giansalvatore Mecca, Paolo Papotti, and Salvatore Raunich. Core schema mappings. In *SIGMOD*, 2009.
- [112] Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, and Marcello Buoncristiano. Concise and expressive mappings with +spicy. *VLDB*, 2009.
- [113] Fabian Nunez. An extended spreadsheet paradigm for data visualisation systems, and its implementation, 2000. Master Thesis, University of Cape Town.
- [114] J.D. Pemberton and A.J. Robson. Spreadsheets in business. *Industrial Management & Data Systems'00*, 2000.
- [115] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 598–609. VLDB Endowment, 2002.
- [116] Lucian Popa, Yannis Velegrakis, Renee J. Miller, Mauricio A. Hernandez, and Ronald Fagin. Translating web data. In *VLDB'02*.
- [117] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a tool for mapping hierarchical schemas. In *SIGMOD*

- '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1271–1274, New York, NY, USA, 2008. ACM.
- [118] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a visual language for explicit schema mappings. In *ICDE*, pages 30–39, 2008.
- [119] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, 2001.
- [120] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
- [121] Vijayshankar Raman and Joseph M. Hellerstein. Potter’s wheel: An interactive data cleaning system. pages 381–390, 2001.
- [122] Vijayshankar Raman and Joseph M. Hellerstein. Potter’s wheel: An interactive data cleaning system. In *The VLDB Journal*, pages 381–390, 2001.
- [123] Frank Rice. Creating XML mappings in excel 2003. Technical report, Microsoft Corporation, 2005.
- [124] Frank Rice. Creating xml mappings in excel 2003. Technical report, Microsoft Corporation, 2005.
- [125] Frank Rice. Introducing the office (2007) open xml file formats. Technical report, Microsoft Corporation, 2006.
- [126] 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.
- [127] 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.*, 45(2):389–409, 2006.

- [128] Barna Saha, Ioana Stanoi, and Kenneth L. Clarkson. Schema covering: a step towards enabling reuse in information integration. In *ICDE*, pages 285–296, 2010.
- [129] Regis Saint-Paul, Hung Vu, Ghazi Al-Naymat, and Boualem Benatallah. Spreadsheet-based complex data transformation. Technical report, CSE-UNSW-0919, 2009.
- [130] Chris Scaffidi, Brad Myers, and Mary Shaw. Intelligently creating and recommending reusable reformatting rules. 2009.
- [131] Christopher Scaffidi, Brad Myers, and Mary Shaw. The topes format editor and parser. Technical report, CMU, 2007.
- [132] Christopher Scaffidi, Brad Myers, and Mary Shaw. Topes: reusable abstractions for validating data. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 1–10, New York, NY, USA, 2008. ACM.
- [133] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.
- [134] Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. Building mashups by example. In *Proceedings of the 13th international conference on Intelligent user interfaces, IUI '08*, pages 139–148, New York, NY, USA, 2008. ACM.
- [135] Jerzy Tyszkiewicz. Spreadsheet as a relational database engine. *SIGMOD'10*.
- [136] F. B. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. Manyeyes: a site for visualization at internet scale. *IEEE Trans Vis Comput Graph*, 13(6):1121–1128, 2007.
- [137] Fernanda B. Viegas, Martin Wattenberg, Jeffrey Heer, and Maneesh Agrawala. Social data analysis workshop, April 2008.

-
- [138] Priscilla Walmsley. *XQuery*. OReilly Media, 2007.
 - [139] Guiling Wang, Shaohua Yang, and Yanbo Han. Mashroom: end-user mashup programming using nested tables. In *WWW'09*.
 - [140] Jeffrey Wong and Jason I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444, New York, NY, USA, 2007. ACM.
 - [141] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW '08*.
 - [142] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. Wysiwyg development of data driven web applications. *Proc. VLDB Endow.*, 1(1):163–175, 2008.
 - [143] Angela Bonifati Zohra Bellahsene and Erhard Rahm. *Schema matching and mapping*. Springer, 2010.