University of Leipzig
Faculty of Mathematics and Computer Science
Institute of Computer Science

The Basics of Complex Correspondences and Functions and their Implementation and Semi-automatic Detection in COMA++.

Master's thesis

Leipzig, September 2011

applied by

Arnold, Patrick
Major M.Sc. Computer Science

Abstract

This Master's thesis describes how a classic schema matcher is enhanced to a sophisticated schema mapper that allows complex correspondences (many-to-many-correspondences) and general functions between two schemas, as well as their semi-automatic detection in addition to the classic (1:1)-correspondence detection. With the latter aspect, this thesis is dedicated to a field in data integration, which has hardly been investigated yet, and offers an approach valuable for many schema matchers. For this purpose, the first part of this study will explain precisely what complex correspondences and functions in the context of schema matching are, how they are connected among each other, what different types exist, under which circumstances they become necessary, and how they are used in real-word scenarios. Throughout the entire first part, examples will be given to underline the meaning and importance of complex correspondences and functions. Besides this, studies and existing solutions about complex correspondences and functions, as well as their detection, will be introduced. In the second part, the actual implementation will be the center of interest, starting with the general extension of the schema matcher COMA++ to represent functions and complex correspondences, and afterwards concentrating especially on the semi-automatic detection of the aforementioned, which plays a key-role in this thesis. At the end, an evaluation will be given to point out the difficulties and benefits of the enhancement, as well as a prospect for future work.

Kurzfassung

In der vorliegenden Masterarbeit wird erläutert, wie ein klassischer Schema Matcher erweitert wird, um Komplexe Korrespondenzen (many-to-many-Korrespondenzen) und allgemeine Funktionen zwischen zwei Schemata auszudrücken, sowie deren automatische Entdeckung als Erweiterung der herkömmlichen Entdeckung von (1:1)-Korrespondenzen. Der letzte Punkt widmet sich dabei einem Gebiet der Datenintegration, das bisher kaum untersucht wurde, und es werden Ansätze vorgestellt, die für viele Schema Matcher eine Bereicherung darstellen können. Zu diesem Zweck werden im ersten Teil der Arbeit Komplexe Korrespondenzen und Funktionen im Bereich des Schema Mappings ausführlich vorgestellt. Es wird gezeigt, wie diese miteinander zusammenhängen, welche Arten unterschieden werden können, unter welchen Bedingungen sie notwendig sind, und wie sie in der Praxis verwendet werden. Im gesamten ersten Teil werden Beispiele gegeben, welche die Bedeutung von Komplexen Korrespondenzen und Funktionen untermauern. Daneben werden Arbeiten und bereits vorhandene Programme vorgestellt, die sich mit Komplexen Korrespondenzen und Funktionen, sowie deren Entdeckung, befassen. Im zweiten Teil steht die eigentliche Implementierung im Mittelpunkt. Zunächst wird dabei die Erweiterung des Schema Matchers COMA++ erläutert, so dass Komplexe Korrespondenzen und Funktionen ausgedrückt werden können. Im Anschluss daran wird das semi-automatische Aufspüren von Komplexen Korrespondenzen und Funktionen beschrieben, was eine Schlüsselrolle in der vorliegenden Arbeit einnimmt. Die Arbeit schließt mit einer Auswertung der implementierten Strategien, welche die Schwierigkeiten und Vorzüge der Erweiterung verdeutlichen sollen, sowie einem Ausblick für zukünftige Arbeiten.

Contents

I hereby declare that this Master's thesis is my own work, and that all references to other works are properly and duly cited. I am aware that any kind of infringement can, even posthumously, entail the revocation of my degree.

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, September 20, 2011 Patrick Arnold

Chapter 1

Introduction

1.1 Motivation

The rise of computerization, internationalization and globalization during the last three decades coined the new term *information society*, a term that is not only used in the wide field of information technology and computer science, but also, or maybe especially, in our everyday life. Today seemingly every person and every company depends on information. A travel agency might cooperate with hundreds of hotels. An airline, which might be cooperating with different travel agencies, offers several flights to various countries. The hotel companies might cooperate with different travel agencies again, and maybe different bus companies will bring the travelers from different airports to their respective hotels. To make sure that these extensive business processes work smoothly, i.e., that the passengers will get to the right airport at the right time, take the right airplane and bus, and finally reach the hotel they have booked, much information is required, and has to be interchanged between different companies. It is quite apparent that information interchange has never been as important as today, and it can hardly be denied that its importance is still highly increasing.

In the age of information and communication technology, many companies and institutions have to deal with a large amount of data, which is to be stored, retrieved, processed and interchanged. It is not very exceptional that many enterprises and organizations, e.g., warehouses, online shops and mail-order companies, travel agencies, transport companies and common carriers, universities, academies and research institutes, as well as financial institutions and insurances have to deal with millions or even billions of information that cannot be handled by humans alone anymore. Today, most enterprises use powerful *database systems* to administer their *database*, i.e., to insert, update, delete, display and process data. As matter of fact, for each database there must exist one unique *schema* that describes exactly how the database is structured, and how it can be accessed by a program or the database system itself.

There are several ways to store data, and even more ways to design a schema. That implies that each company has its own database schema, which normally differs considerably from other database schemas, even if both schemas represent the same kind of information. This phenomenon is called *heterogeneity*, and will play a big part in this thesis, because it is the actual reason for schema matching.

Now difficulties arise when two or more companies incorporate, or at least cooperate, and therefore decide to create a new database on the basis of the existing ones. In this case, either a new schema is to be developed, or one of the existing schemas will serve as the schema for the new database. Either way, *schema mapping* is inevitable then, i.e., rules must be declared that explain how data of the old database can be translated into the new database. For this purpose, *schema matchers* help to detect the relations (links) between the schemas so that those rules can be declared. After this is accomplished, the original databases can be transformed to the

target database, which is called *data transformation*. The entire process, so schema matching, schema mapping and data transformation, constitutes a crucial part in the wide field of *data integration*, and plays a very big part today; not only since so many enterprises and research companies are obliged to merge their databases, but also due to the steady rising of customer's requirements. Customization, individualization and a large range of products are nowadays very important criteria for most customers. The rise of the world wide web led to the fact that customers can acquire products and even services all over the world so that most enterprises are facing an enormous competitive pressure today. Product catalogs have to be changed almost daily, new branches have to be opened, others to be abandoned, and others again to be merged or split. Thus, a company can be occasionally forced to considerably change its database or even to develop a new one, especially when the original database was not designed very well, or, as it often happens in information systems, was gradually extended over the years until it was not possible to manage it any longer. In such a case, data integration can be necessary again, that is, an old database has to be mapped to a new one.

The result of the schema matching process is called a *map* or *mapping*, and can be seen as a list of links (relationships) between the different elements of the two (or more than two) schemas. Such a link is called *correspondence*, and could be, for instance, something like $(DB1.Address,\ DB2.PlaceOfResidence)$, meaning that the elements *Address* in Database 1 and $PlaceOfResidence$ in Database 2 express the same kind of information. Having this map, there are different possibilities of what can be done next. For example, a data transformator could transfer the instances of $DB1.Address$ to $DB2.PlaceOfResidence$, or a merging tool could get the list of correspondences to merge the two schemas.

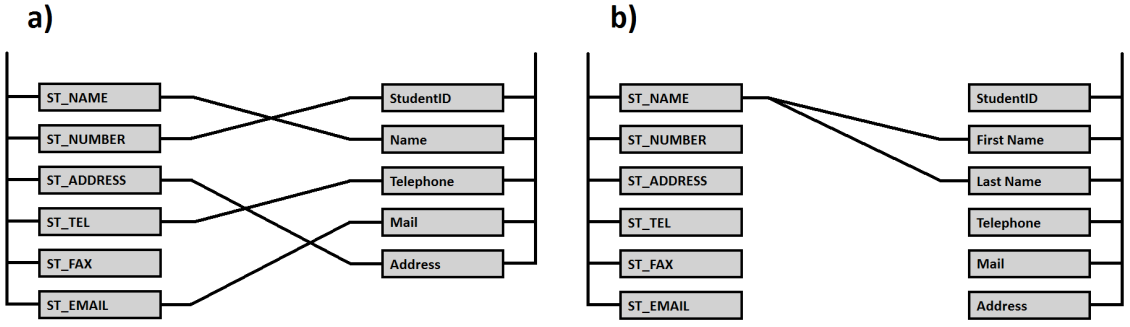


Figure 1.1: **a)** A typical map containing only (1:1)-correspondences. **b)** A map containing a (1:n)-correspondence (other correspondences omitted).

A rather simple map would be a set of (1:1)-relationships between the different elements of the respective schemas. Figure 1.1 a) shows such a simple example. It is certainly appropriate to demonstrate schema matching in general, however, in practice schemas are normally much more extensive, possibly having hundreds of elements. If such large schemas are to be matched, (1:1)-relationships will not be

sufficient anymore. (1:n)-relationships and (n:1)-relationships will occur frequently, and even the possibility of (m:n)-relationships cannot be entirely excluded.

Figure 1.1 b) shows a more sophisticated example with a (1:n)-relationship. Such a *one-to-many-relationship* requires a function to manipulate the data so that it will be consistent with the new database. The function has to split each name from the source database into its first name and last name so that it matches the schema of the target database. Such a function could be seen as the typical split (or substring) function, which is well-known in programming languages and theoretical computer science. In fact, it will be shown in this thesis that functions used in schema mapping are rather similar to functions used in programming languages.

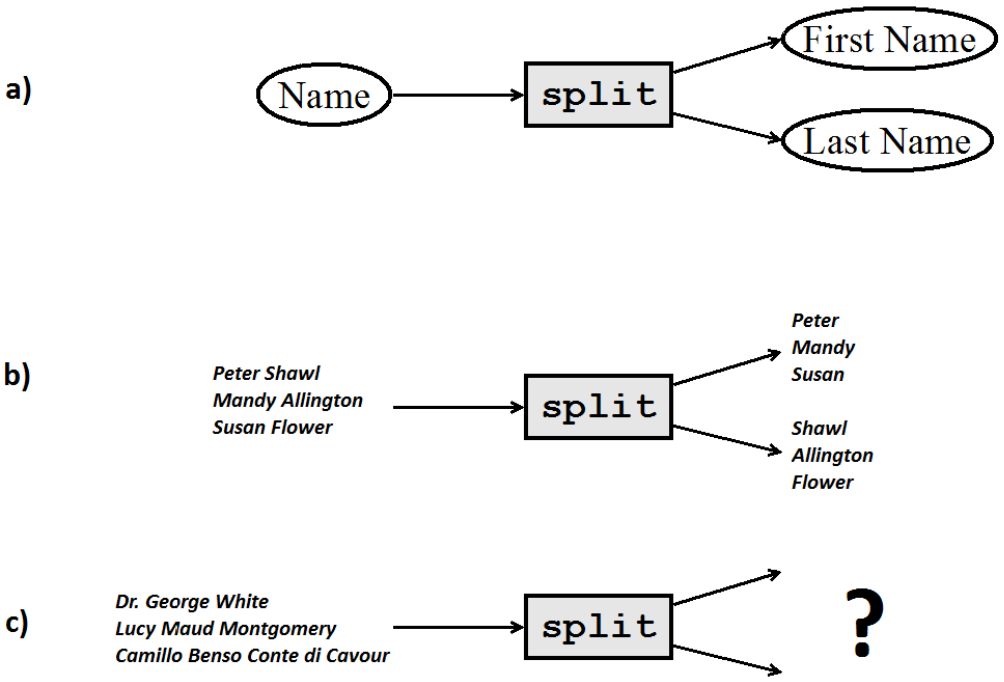


Figure 1.2: Using the split-function for the complex correspondence (Name, (First Name, Last Name)). The picture depicts that it works for common names (b), but immediately fails if the name consists of more than two words (c).

Figure 1.2 demonstrates the usage of the split function so that the names from the left-hand database match the schema of the right-hand database. The example shows that names like "Peter Shawl" could be easily processed by the split function, however, as soon as it comes to persons having a title or more than one first or last name, it would be difficult to extract the correct first name and last name. This is a major problem of using functions – they cannot be executed if the instance data does not match its specification. In case of transformation, the instances shown in picture c) would be either rejected or translated incorrectly.

Of course, there are many other functions which might be needed in the context of schema mapping as well, such as arithmetical functions to add values together or to compute the average, minimum or maximum of a set of values. There are also many significant functions for (1:1)-relationships, such as functions that convert a number into a string, an US date format into a German date format, or a common string into a lowercase string. Such functions are especially used to warrant high data quality.

Functions play a big part in schema mapping after all, but unfortunately there has not been much research about it yet. While the detection of matches between two elements has always been the centerpiece of many approaches and studies, complex matches and their need for functions has often been completely disregarded.

In this Master's thesis, schema matching and mapping will be the focus of interest, however, the emphasis will be on complex correspondences between schemas, and particularly on functions. It will be explained what functions in the context of schema mapping are, where and how they are used, which important functions exist, how they can be classified, and especially how they can be detected automatically, which will play a key role in the second part of this study. Schema mapping applications will be scrutinized, especially with regard to their function sets, and scientific approaches about many-to-many-correspondences will be presented and explained, even though functions and complex correspondences in the context of schema mapping have not been examined much so far.

Finally, the central part of this thesis will be the implementation of functions, function detection and complex correspondence detection in COMA++, which is something no schema matcher can entirely offer yet. Hence, it will be explained elaborately how functions have been implemented in the new schema mapping project developed at the Web Data Integration Laboratory, University of Leipzig, where this Master's thesis was initiated, supervised and supported.

1.2 About the WDI Lab

The Web Data Integration Laboratory [27], normally abbreviated as WDI Lab, is a research laboratory of the Institute of Computer Science, University of Leipzig, where new technologies for data integration and data transformation are developed and implemented, especially with regard to enterprise data and web data. In 2009, the Federal Ministry of Education and Research contributed 1.5 million Euro to establish new strategies and solutions for the purpose of web data integration [6]. This new software is partly based on COMA++ [1], a schema matching program

which has already been developed at the Institute of Computer Science since 2002, and which got much positive feedback ever since.

One task has been to extend COMA++ by the possibility of dealing with complex correspondences and functions to grant more sophisticated and more reliable data integration. In the context of this Master's thesis, these features were implemented, and will be explicitly explained in the fourth chapter, which will roughly make up half of the thesis.

The solution that was developed in the context of this Master's thesis was a first version, which was later integrated in the COMA++ software. In this study, the term "first version" will therefore often be used. In later years, an extended version might be developed, which of course is not certain at the moment.

1.3 Contributions

This Master's thesis includes the following aspects:

- Analysis and assessment of existing schema mappers, especially with regard to functions and complex correspondences.
- Analysis of existing studies about complex correspondence detection and function detection.
- Improving COMA++ by offering complex correspondences and functions.
- Development of function detection algorithms and complex correspondence detection algorithms.
- Adjusting the data transformation module of COMA++.
- Tests, evaluation and documentation.

The part of the data transformation is omitted in this thesis, but was an important step for the new COMA++ release nonetheless.

1.4 Outline

The second chapter concentrates on schema matching and mapping in general. It will be explained what schemas, ontologies and correspondences are, what kind of correspondences exist, and what the difference between schema mapping and schema matching is. As for the functions, an entire section is dedicated to this topic, including many examples to underline the meaning and importance of functions in schema mapping.

In the third chapter, a couple of schema mapping and schema matching programs will be briefly presented and compared with each other, and a few studies concentrating on automatic function detection will be introduced, even though there are not many approaches about complex correspondence and function detection available.

The second and third chapter are the base for the fourth chapter, where the implementation of functions, function detection and complex correspondence detection in the new COMA++ project will be eventually explained. The main focus will be on automatic generation of functions and complex correspondences, though, because this aspect is certainly the most interesting and most sophisticated one. Besides this, an evaluation will be given to assess how good the implemented algorithms work after all, and on which parameters they depend.

Finally, this thesis concludes with a summary, where the most important aspects and results of the study will be condensed.

Chapter 2

An Introduction in Schema Mapping

2.1 Data Integration

A central problem of *data integration* is the heterogeneity between two different databases, each having its own specific schema, that are in any way to be integrated, e.g., have to be merged to one logical (yet not necessarily physical) database. Of course, there must exist exactly one schema for this logical database, which is normally called *target schema*.

Data Integration encompasses three significant steps:

- Schema Matching
- Schema Mapping
- Data Transformation

Schema matching is always the first step when data integration is to be performed. Its result can be seen as a set of links which explain how schemas correlate, i.e., how they are connected and how data objects must be mapped to the target schema so that the final database is consistent and no information gets lost. After this, *schema mapping* takes part. In this step, transformation rules are declared so that a script or program can translate data objects from the source database to the target database. The process of translating data objects is called *data transformation* or data translation; hence, data objects are copied to the final database, either at one time or at runtime. In the context of data transformation, *object matching* plays a big part. Since it is always possible that a unique data row appears in different databases at the same time, object matching is necessary during or after data transformation to avoid redundancy and inconsistency.

Schema mapping is inevitable both for *physical data integration* (e.g., data warehousing) and *virtual data integration* (e.g., federated database systems). In case of virtual data integration, all data objects remain in their original database, and will be accessed by a mediator at runtime. The mediator therefore receives queries from a user, and then using the map, accesses the different databases to retrieve the matching data sets. For the users it seems as if they interact with just one database, and for this reason these different databases are considered a logical database. However, in the context of physical data integration, all data objects are immediately transferred to a target database after schema mapping has been performed, and the user is able to directly interact with this database. In this case, the data transformer has to use the map to copy the data objects properly into the target database.

This Master's thesis concentrates on schema matching, and to a less degree on schema mapping. It will be explained more precisely in the next sections.

2.2 Basics of Schema Mapping

2.2.1 Schemas, Taxonomies and Ontologies

Each *database* requires a unique *database schema*, which is used to describe the structure of the database [13]. The schema itself is described in a specific *DDL* (data definition language), and can be depicted as a graph or a set of tables in case of a relational database. In schema matching, graphs are normally preferred.

There are different kinds of schemas used to define a database. In many cases, the following layers are regarded [13] [30]:

- Physical Schema (Internal Level)
- Conceptual Schema (Conceptual Level)
- Logical Schema (External Level)

The *physical schema* describes the physical and technical view on the database. It especially concentrates on index structures, the way how data is stored, and the way how it can be accessed by the database system [22]. The *conceptual schema* describes the complete structure of the database, yet unlike the physical schema does not regard system-specific properties. Finally, the *logical schema* is derived from the conceptual schema, and describes a subset of the database which applies to a specific user or application. The reason for using logical schemas is that usually each application connected with the database does not have full access to the database. The logical schemas therefore define which part these applications may use, and how data visible to the application can be accessed. As described in [22], logical schemas can also be seen as an interface between database and applications.

Thus, a database may be defined by an internal schema, a conceptual schema and a set of external schemas, which are the interface to the several applications. Schema matching typically concentrates on the conceptual schema, so in the following the term *schema* will always refer to the conceptual schema, only describing the structure of the whole database.

Ontologies are widely used in the field of semantic web, and are a collection of terms as well as their relationships amongst each other [26]. Schemas and ontologies are often used in the same context, however, schemas and ontologies (resp. schema matching and ontology matching) are not the same. An ontology typically possesses a vocabulary that describes a special domain of interest [8], as well as a specification which describes the meaning of the terms in the vocabulary. According to [8], an ontology comprises several data models or conceptual models, i.e., a database schema can be seen as a subset of an ontology, or in other words, an ontology represents more than a database schema. While a database schema only describes the structure of a database to access data by a database system, an ontology describes both the structure and the meaning, so it also considers the semantic aspect.

A *taxonomy* is a hierarchical classification of terms, and is therefore a subset of ontologies. Taxonomies used by (online) stores and mail-order companies are also called *catalog*.

Since taxonomies have no loops, they seem rather similar to schemas. However, when it comes to schema matching, there is a large difference between schemas and taxonomies. While schemas describe the structure of real-world objects, which are to be represented according to this schema, a taxonomy only describes where such an object belongs within a large system. For instance, a schema describes how a book is to be represented. Books may have a title, an author, a year when they were published, a genre and an annotation. Now a taxonomy could be used to classify the several book objects. It might differ between fiction and non-fiction, classic and modern literature, adventure, romance, fantasy and thriller literature, children, young adult and grown-up literature, etc.

Even though there are differences between schemas and ontologies, schema matching and ontology matching (resp. mapping) are rather similar after all. In this Master's thesis, only schemas will be the focus of interest, but most statements about schemas will also apply to ontologies. One main difference between schemas and ontologies is, however, that ontologies can be cyclic whereas schemas are invariably acyclic.

Schemas can be either flat or hierarchical. Relational database schemas are typically rather flat, because the first normal form is highly recommended when designing a database schema, and nested structures are therefore not allowed. In this case, the schema would consist of at most two levels: tables and attributes (columns). Possibly there could be a third level though, containing meta-data like data type, domain, foreign key, etc. On the contrary, XML schemas are often quite hierarchical, having several levels on which data might be stored.

Trees are widely used in schema matching to represent a schema. In such a tree, the several items are normally called *nodes* or *elements*. As known from graph theory, nodes that have no subnodes are called *leaves*, and all other nodes are called *inner nodes*. Nodes which have no father are called *root*, and in each tree there exists exactly one root.

There are different ways to store data persistently: in a relational or object-oriented database (using a database schema), in an XML file (using an XML schema), or in a single file or a set of files (e.g., csv files, which often have no explicit schema). The database which is conform to a given schema is also called an *instance* of this schema, the files in which this data is stored are called *instance files*. An instance may consist of $1 \,..\, n$ (n finite) data sets, which are also called *record* (especially in XML files), *tuple* or *data row* (especially in relational databases).

2.2.2 Mapping and Matching

As mentioned in the first chapter, schemas are normally most different, even if they describe similar databases. The difference between two schemas is called *heterogeneity*.

Given schemas S_1, S_2, ..., S_n ($n \geq 2$) which are heterogeneous and represent databases DB_1, DB_2, ..., DB_n, one crucial step in *data integration* is to project ("map") those schemas to a schema S_T, which will serve as the global schema for the databases DB_1, DB_2, ..., DB_n. This process is called *schema mapping*. Another aim could be to generate a schema S_T out of the existing schemas S_1, ..., S_n, which would then compromise every aspect described in those schemas. This process is called *schema integration*, the result is called *integrated schema* [25].

The process of detecting the links between those schemas, e.g., ($Student.Address$, $Std.PlaceOfResidence$) is called *schema matching*. Schema matchers help the user to find those links, but many classic schema mappers do not offer this feature. This is because schema matching and schema mapping are rather different things, although they are both important aspects of data integration.

S_1, S_2, ..., S_n are called the *source schemas* and S_T the *target schema* or *destination schema*. Normally, there is only one target schema, but there can be any number of source schemas. For simplicity, only 2 schemas will be assumed in this study, so a source schema and a target schema.

The main task of schema mapping is generating queries that express how data transformation is to be performed. The queries are generated out of the accomplished map, and are stored in a transformation script. Later on, a program can execute this script, and this way perform data transformation.

There are different standards to specify such a transformation specification. Many programs like [17] or [23] generate XSLT scripts or XQuery scripts which specify how data is to be transformed. Another possibility is to declare SQL statements that carry out the transformation process [25].

Of course, it is also possible to add a complete transformation engine to the schema mapper so that the program does not only map schemas, but also performs the transformation itself. That is the way how COMA++ will work, and in this case a separate language could be developed as well.

It has to be noted here that the terms *schema matching* and *schema mapping* are often used in different ways, and sometimes are even considered synonymous. In this thesis, matching will refer to the process of detecting links between schemas, whereas mapping means to build transformation scripts which allow a program to perform data transformation. However, these two aspects are interrelated, because schema matching alone would be useless if no schema mapping would be performed afterwards, and before schema mapping can be done, schema matching had to be performed (whether it was done manually or automatically).

2.2.3 Heterogeneity

As mentioned before, heterogeneity is the center of interest in schema matching, because only heterogeneity between source schema and target schema makes schema matching even necessary. *Heterogeneity* can be seen as a set of conflicts between two or more than two databases, which are especially caused due to the usage of different database systems and data models, different requirements and different schema structures [25]. The latter point is the main problem in schema matching and mapping.

There are different conflict categories as enumerated in [25] and [4]:

- Technical heterogeneity
- Syntactical heterogeneity
- Heterogeneity between data models
- Structural heterogeneity
- Heterogeneity between schema elements
- Semantic heterogeneity

Technical heterogeneity refers to differences between technical aspects, such as different ways of data retrieval and different query languages, different communication protocols (like HTTP and JDBC) and different database interfaces. *Syntactical heterogeneity* is quite similar and means differences in the presentation. This involves different character encoding and different binary number formats.

Heterogeneity between data models occurs if two different data models are used, for example, a relational model and an XML model [25]. Since each data model has its own features and qualities, different data models normally involve structural and semantic heterogeneity.

Structural heterogeneity encompasses structural differences between two schemas. Even though two schemas express the same thing, they can be designed in many different ways. For example, the relation *Address* in two databases could be represented as (*addr_id*, *street*, *zip*, *city*, *country*) or as (*addr_id*, *street*, *number*, *zip/city*, *country*). Heterogeneity between schema elements is a subset of conflicts that belong to structural heterogeneity. In this case, different elements of the data model are used to represent the same thing. For example, in a relational database the address of an employee could be a table or an attribute of a relation.

Semantic heterogeneity refers to the meaning of elements in the schemas. Important examples are synonyms (different words which have the same meaning, e.g., car and auto), hypernyms (words which include other words, e.g., vehicle is the hypernym of auto) and homonyms (words which have different meanings, like mouse, coach, etc.). Semantic heterogeneity also includes different measure units (like Fahrenheit and Celsius) and different scales.

In schema matching, the technical and syntactic heterogeneity are normally disregarded, because they are of no use in the context of correspondence detection. For this, the problems of structural and semantic heterogeneity are most important, and will play a big part in this study.

2.2.4 Correspondences

A *correspondence* is a relationship between a set of source nodes and a set of target nodes. Correspondences are also called *match*, especially when they were detected by a program. A correspondence means normally a true relationship, whereas programs consider all links that they have detected as a correspondence. As matter of fact, such correspondences may be wrong (false matches), meaning that the user has to remove them, or to declare them invalid to get a correct mapping.

There are different perspectives on correspondences and how they can be classified. In this section three perspectives will be regarded:

- The cardinality
- The kind of relationship
- The kind of source and target element(s)

Cardinality

The cardinality of a relationship can be (1:1), (1:n), (n:1) and (m:n), but the latter case is rather seldom. In a (1:1)-correspondence only one source element and one target element take part. In a (1:n)-correspondence one source element and more than one target element take part; the definition of an (n:1)-correspondence is analogous. In an (m:n)-correspondence there are more than one source and more than one target elements. Hence, this definition states that (m:n)-correspondences do not include correspondences of the type (1:1), (1:n) and (n:1).

Correspondences are always directed, because data from the source databases has to be transferred to the target database, but never vice versa. Therefore, (1:n)-correspondences and (n:1)-correspondences cannot be seen as equivalent, even if it looks enticing to do so. As it will be shown in the second part of this thesis, there exists indeed a great difference between those two kind of relationships.

Correspondences which are not (1:1)-relationships are also called *complex correspondences*, also indicating that such a relationship between schema elements is much more complex and more difficult to detect than (1:1)-correspondences [15].

(1:n)-correspondences and (n:1)-correspondences must never be considered as a set of (1:1)-correspondences, even though it might appear so when beholding a map.

Truth is that schema matchers occasionally return several (1:1)-correspondences like ($FirstName$, $Name$) and ($LastName$, $Name$), which are also called *local (1:1)-correspondences*, but actually mean a global (n:1)-correspondence, that is (($FirstName$, $LastName$), $Name$). These global correspondences are the only correct ones after all.

For example, let (($Name$), ($Title$, $FirstName$, $LastName$)) be a (1:3)-correspondence. This correspondence may not be seen as three (1:1)-correspondences, because this complex correspondence is not equal to the 3 local correspondences ($Name$, $Title$), ($Name$, $FirstName$) and ($Name$, $LastName$), but $Name$ refers to $Title$, $FirstName$ and $LastName$ at the same time. In a complex correspondence like this, there still exists only one relationship between the set of source and target elements, whereas in the case of three (1:1)-correspondences three separate relationships exist.

A complex correspondence is therefore a set of source elements and a set of target elements (both not empty, and at least one of them containing more than one element) with exactly one relationship in between.

Complex correspondences are mostly (1:n)-correspondences or (n:1)-correspondences. Apart from the classic "name"-example, there are several other examples, e.g., $Address = Street + ZIPCode + City + Country$, or $BookPrice = Price * (1 + SalesTax)$, or $EmployeeID = FirstLetterOf(FirstName) + FirstLetterOf(LastName) + DateOfBirth$.

On the contrary, (m:n)-relationships are very rare. An admittedly somewhat peculiar example would be the following: company A represents an address using two elements: the first one represents street and zip code, and the second one represents city and country. Company B represents the address by the three elements $Street$, $ZIP/City$, $Country$. This correspondence would be therefore a true (m:n)-correspondence, i.e., (($StreetZip$, $CityCountry$), ($Street$, $ZipCity$, $Country$)). However, this is a special case which will hardly occur in schema matching, because nobody would seriously design such schemas (especially such a source schema).

It must be remembered that a correspondence between ($Street$, Zip, $City$, $Country$) and ($StreetZip$, $CityCountry$)) would not be an (m:n)-correspondence, even though it rather looks like one. In fact, it is a set of two (n:1)-correspondences: (($Street, Zip$), $StreetZip$) and (($City$, $Country$), $CityCountry$).

As described in [4], the cardinality of correspondences and mapping types can be subdivided more precisely. The following 11 mapping types are distinguished: (1:0), (0:1), (1:1), (1:C), (C:1), (C:B), (1:N), (N:1), (C:N), (N:C), (N:M), where C and B are constants (values that never change), and (1:0) and (0:1) means that there is no partner for the source element resp. target element. However, only the four classic types will be considered in this study.

Kind of relationships

A correspondence can represent different kind of relationships. Let X be an element of the source schema and Y be an element of the target schema. There are four possible kind of relationships between those two elements [22]:

- Equivalence ($X \equiv Y$, also called *equal*)
- Inclusion ($X \subseteq Y$ or $Y \subseteq X$, also called *is-a* or *inverse is-a*)
- Overlap ($X \cap Y$)
- Disjunction ($X \cap Y = \emptyset$, also called *is-not*).

In schema matching, the focus is often on equal-relationships, and to a less degree on is-a and inverse is-a relationships. The disjunction relationship is normally of no use in case of schema mapping, so it can be presumed that each relationship is of the first, second or third kind. When dealing with large and very heterogeneous schemas, it often happens that the overlap and disjunction type prevail, and real equivalences occur rather seldom, so the more heterogeneous schemas are, the less equivalence relationships can be expected.

Kind of Source and Target Element(s)

In the following, let us assume that a schema is represented by a tree, and that the values are always stored in the leaf nodes. There can be four different types of simple correspondences with regard to the node type.

1. Both source node and target node are inner nodes.
2. The source node is an inner node, yet the target node is a leaf.
3. The source node is a leaf, yet the target node is an inner node.
4. Both source node and target node are leaves.

When performing data transformation, only the last case is to be regarded, because in case 1 and 2 no data can be fetched at all, and in case 3 the data objects of the source node would be transferred to an inner node of the target schema, where it cannot be stored. A true correspondence is therefore always between two leaves (case 4), but schema matchers often produce correspondences of all cases. However, this does not pose any problem at all, because those correspondences, though false, can be very helpful to find new correspondences or the correct correspondence respectively.

According to [25], correspondences of type 1, 2 or 3 are called *high-order correspondences*.

2.2.5 Map

A list of correspondences is called an *alignment* or *map*. The map is therefore the result of the matching process, and will be the input for the mapping process.

Mostly the terms alignment, map, mapping and "list of correspondences" are considered synonymous, where the term alignment is rather used in ontology matching and map rather in schema mapping. In this thesis, the word *map* is used, but mapping and map express the same thing.

2.3 Functions

2.3.1 Introduction

Functions play a big part in schema matching and mapping, because the correspondences themselves only indicate the relations between two schemas, but do not always warrant a correct data transformation. Correspondences specify which elements between different schemas belong together, but normally each database schema specifies several constraints for its elements, e.g., a special data type, a special number format, a domain, etc. In fact, today's database systems are powerful enough to apply a huge amount of different constraints to each database element, which often makes it necessary to adjust data sets before they are transformed between different databases.

To enable a sophisticated data transformation, there exist many functions which can be used in addition to the correspondences. In this section, functions will be explained more precisely, and examples and classifications will be given. It will also be shown that functions are necessary to build complex correspondences.

Figure 2.1 shows a map in which functions are used. The price in schema 1 is to be converted into another currency (e.g., US-Dollar), so it is first multiplied by the exchange course of 1.29 and then rounded. Besides, the tax price is increased by 2.5.

While correspondence detection is a classic task of schema matchers, functions rather belong to the schema mapping process, i.e., they are declared to ensure proper data transformation. The data transformator will execute these functions to adjust the data objects so that they fit the target schema. However, since complex correspondences always require a function, it might also be possible to consider function detection as a part of schema matching.

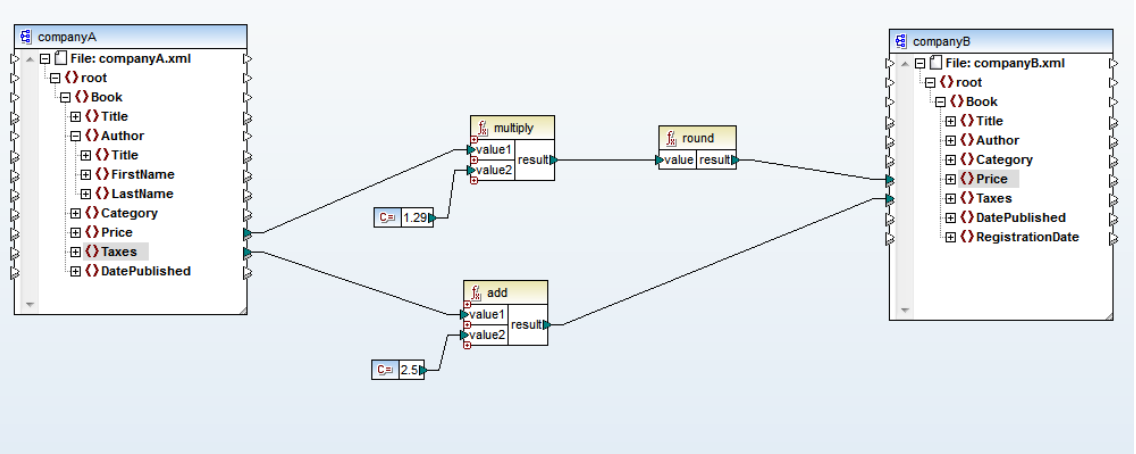


Figure 2.1: **a)** A map utilizing functions for different correspondences, performed in Altova MapForce 2009.

2.3.2 Definitions

Functions in schema mapping are not quite different compared to mathematical functions or functions well-known in programming languages, however, there exist several definitions of functions.

In algebra, a *function* is a relation f between two sets X and Y, where for each $x \in X$ exists a $y \in Y$ so that $(x, y) \in f$, and for each $(x, y) \in f$ and $(x, z) \in f$ is $x = z$ [32]. X is also called the domain and Y the codomain of f.

In the field of software engineering, the definition of functions is less formal than in algebra. In [26] functions (procedures) are described as subprograms (subroutines) containing an algorithm that is employed to solve a subproblem of a superior problem. In this case, functions are especially used for the sake of structuring and reuse.

Either way, a function can be seen as kind of a black box which obtains a set of parameters, does something with these parameters, and returns a unique value. However, for the subject of this Master's thesis neither of the presented definitions seems to be much satisfying, especially with regard to the implementation of functions in COMA++. Therefore, a separate definition is given now and will be henceforth used.

A function in schema mapping is defined as a triple (f, P, r), where f is a unique name, P a list of parameters and r a return value.

The list of parameters P contains Parameters $p_1,\ p_2,\ ...,\ p_n$, $(n \geq 0, n$ finite), where each p_i is either mandatory or optional, and can be the value of a source element, a constant value or the return value of another function. Also, to each parameter

there exists a specific set of allowed data types; in most cases, a parameter must be of one unique data type, such as string, integer, float, double, date, etc.

The return value r is of a specific data type as well, and can be mapped to an element of the target schema, or to another function where it would serve as a parameter then. Even though each function returns exactly one value, this value can be mapped to more than one element or function.

A parameter can be a source node reference, a constant or the return value of another function. As matter of fact, a correspondence can have any number of functions. Normally, functions are part of a correspondence, refining this relationship to deal with complex correspondences or to allow a sophisticated data transformation. However, there exists a special case where a function is not part of a correspondence. This case occurs if a single function has no parameters or only constants as parameters.

Example 1. The function $add(salary,\ commission) = salary$ has two parameters which are both source node references. The *add* function is one of the few functions, where the order of parameters is of no importance, and where an arbitrary number of parameters is admissible.

Example 2. The function $concat(firstname,\ "\ ",\ lastname) = name$ has three parameters: a source node reference, a constant, and a source node reference. Again, the number of parameters is unbounded in this function, but the order is now of importance.

Example 3. The function $currentDate() = registration\_date$ has no input parameters at all. If used alone, this function will never be part of a correspondence, because no source element can point to it.

Note that the function definition given above is not a strictly mathematical definition, because its scope is just to point out the general concept. We discuss and explain this point in more detail in chapter 4.

2.3.3 Examples

The functions *concatentation* and *split* are very common functions, especially when it comes to complex correspondences. For instance, in schema A the name of an employee is represented as $Title + FirstName + LastName$, but in schema B only by the element $Name$. In this case, the concatenation function is necessary to put $Title$, $FirstName$ and $LastName$ together, and to put space characters between those subelements of schema A to match the $Name$ element in schema B.

In some cases, values of an element are declared to be lowercase or uppercase. For instance, national abbreviations are often uppercase values (e.g., D, UK, CH),

whereas country-specific top level domains are often lowercase (e.g., de, at, ru, uk). However, one cannot expect that all values of a source database oblige to the aforementioned rules, and in this case the *uppercase* and *lowercase* function might be helpful to warrant that all values are indeed lowercase resp. uppercase after data transformation.

Sometimes string values might have undesirable whitespace characters at their beginning or end, maybe as a result of sloppy data extraction from a csv file. Then the *trim* function might be useful to remove these blanks.

The *replace* function (sometimes called translate) can be very useful, too. For instance, when a German-language database must be mapped to an English-language one, it could be required that special characters like ä, ö, ü and ß are to be replaced by ae, oe, ue and ss. Of course, there are also different dialects of English (e.g., British English and American English), so it might be desired to replace words like colour, schemata and catalogue by color, schemas and catalog, or abbreviations like Dr, Mrs, Prof, etc. by Dr., Mrs., Prof., etc. When companies incorporate, a unique terminology is often required, so the replace function can be also used to warrant such a terminology. For example, both schema A and schema B might have an element $Profession$, yet for some professions exist different terms like *vendor* and *salesman*, *adviser* and *consultant* or *joiner* and *carpenter*. When those databases are merged, replace-functions like $replace(\text{"}vendor\text{"},\text{"}salesman\text{"})$ can be very helpful again.

Arithmetic functions are often necessary as well. For instance, schema A has an element $TotalPrice$, which consists of the subelements $Price$ and $SalesTax$. Schema B, however, has only the $Price$ element, and therefore the functions *add* and *multiply* would be required: $B.Price = multiply(A.TotalPrice.Price, add(1, A.TotalPrice.SalesTax))$. There are many other cases where mathematical functions seem inevitable, for instance, when dealing with different measure units or currencies. Sometimes values should be rounded, for instance, if a company that sales cars might desire that their car prices will be integers (e.g., 7399 Euro instead of 7399.80 Euro). When this company merges its database with other databases where car prices can be rational numbers, functions like *round*, *floor* and *ceiling* might be used.

Functions like sum, average (avg), maximum (max) and minimum (min) are called *aggregation functions*, but often belong to the arithmetic function group. They are used to create a single value out of different values that belong to exactly one element. In a large bookstore, there could be several copies of a book, each published by a different publisher. Hence, there might be different prices for this book. When bookstores merge their databases, they might decide that only the cheapest book will be offered in future, and therefore have to decide for each book which publisher they choose. A function like $min(bookPrice)$ would be helpful then to realize this, meaning that from a set of copies only the cheapest one is to be mapped. In this case, the aggregation function seems rather like a filter, but functions like sum or avg are often no filter functions.

Conversion functions are only necessary if the two schemas of a map specify data types for their elements, and if the data types between two elements of a correspondence are not the same. In many schemas, especially in XML schemas, data types are often neglected or constantly declared as strings, but if different data types occur in a correspondence, using the respective conversion function is rather recommendable.

Date functions can be widely used as well. The different date formats that could appear between schemas is a main problem when dealing with date elements, e.g., November 23, 2010 could be represented as 2010-11-23 or 11-23-2010 or 11/23/2010 or 23.11.2010, but also as Nov 23, 2010, etc. It would be difficult to convert date formats only by use of split-, concat- and index-of-functions, so date functions which are able to convert common date formats directly seem very convenient.

It should have been pointed out that there are many examples where functions between elements of a correspondence are necessary. There are many other examples, of course, yet the rather simple examples presented above were also supposed to demonstrate that even between very common and simple schemas the use of functions is rather significant.

2.3.4 Functions and Complex Correspondences

In case of an (n:1)- or (1:n)-correspondence, using functions is inevitable. It can be even said that in this case each source element must participate in at least one function, or otherwise the correspondence would be redundant. If no function is used at all, the correspondence would be even a logical (1:1)-correspondence. This seemingly bold statement has the following reason: if source elements x_1, x_2, ..., x_n are mapped to the target element y, x_1 would be mapped to y, but then x_2 would be also mapped to y, and thus would overwrite the former value in y, which was x_1. Then, x_3 would overwrite x_2 and so forth. This would proceed until x_n is mapped to y, and thus x_n would always be the value which is stored in y when the data transformation engine has finished parsing this correspondence, so the alleged (n:1)-correspondence is always a correspondence between x_n and y, so a (1:1)-correspondence.

The opposite case is a little different, because when a (1:n)-correspondence does not use any function, nothing will be overwritten when transforming data sets, and it might almost appear that in this case the complex correspondence does not necessarily need a function at all. However, in this case the source element x would be simply mapped to each target element y_1, y_2, ..., y_m ($m > 1$), so in this case the alleged (1:n)-correspondence would be a set of (1:1)-correspondences, and as already explained in section 2.2.4, a set of (1:1)-correspondences is not a complex correspondence.

In this thesis, functions that are used to enable complex correspondences are called *connector functions* for a better distinction between general functions and functions only needed to express complex correspondences. Basically there are two kinds of connector functions: concatenation and split for textual elements, as well as arithmetic functions like add, subtract, multiply and divide for numerical functions. Complex correspondences between dates, e.g., (*date* (*month*, *day*, *year*), *date*) can be handled by concatenation and split functions or by a special date converting function.

2.3.5 Classifications

Functions are mainly classified by the data type that the function gets as its (main) parameter; the return value is often of this data type as well. Such data types are especially string, number and date. However, there are also functions which cannot be assigned to a special data type, such as logical and relational functions. The following classification is oriented towards the classification used in Microsoft BizTalk Server [17].

In this thesis we consider the following 7 *function groups* (also called functoid in Microsoft BizTalk Server):

- Numerical functions
- String functions
- Date and time functions
- Conversion functions
- Logical functions
- Relational functions
- Special functions and advanced functions

Numerical functions are sometimes also called mathematic functions or arithmetic functions, and comprise functions like sum, subtract, multiply, divide, modulo, logarithm, (square) root, sinus, cosine, etc. Their input parameters and return value are always numbers.

String functions are used to manipulate and analyze strings. Typical string functions are concatenation ("concat"), split (substring), trim, lowercase, uppercase, left-trim, right-trim, replace (translate), index-of (search), etc. Their input and output are often strings.

Date and time functions deal with dates and times; they are normally used to get the current time or date, or to convert a date into another date format. Their parameters and return values are often dates.

Conversion functions are used to convert the data type of an element into another type, e.g., from integer to string or vice versa.

Logical functions like and, or, not, and *relational functions* like equal, greater-than, less-than, etc. are only used for filters. Filter functions enable to create filters that specify under which circumstances a data row is to be mapped, or to which position in the target schema a data row is to be mapped. Some programs allow to create filters by using functions, others have a separate filter function. Filters will be explained more explicitly in section 2.3.8.

Special and advanced functions are functions which are too specific to fit in any other function group. This functoid could comprise very special numerical functions, such as hexadecimal-to-decimal or is-prime, etc. Also computer-specific functions like ASCII-to-decimal, and functions needed for filters like if-then-else, switch-case, loop, etc. belong to this group.

2.3.6 Importance

The examples presented in section 2.3.3 should leave no doubt that functions are pretty significant in schema mapping, and constitute a great advantage to every classic schema mapping program. However, there are apparent differences between the importance of the several functions, and even between the several function groups. Some functions have a high importance, because they might be frequently used, whereas others seem rather unimportant and are scarcely used. Indeed, the importance of functions appears to follow Pareto's principle, meaning that about 20 % of all functions (or even less) cover 80 % (or even more) cases in which functions are needed [31].

When functions are to be implemented in an existing schema mapper or matcher, it should be of great interest which functions are the most important ones, so which are to be implemented first.

The following functions are certainly of high importance:

- Arithmetic functions (sum, subtract, multiply, divide)
- concatenation and split
- trim, uppercase, lowercase
- replace
- current-date, current-time
- conversion functions, date format converting functions

Some programs offer up to 100 functions, but as shown above, only a few functions

are really of high importance. Functions like logarithm, sinus, cosine, index-of, ASCII-to-Decimal, etc. are only in some very specific contexts useful.

There are also differences in importance between function groups, yet to a less degree. It seems that string functions are used very often, and might have the widest usage in many cases. However, the usage of functions depends much on the context and the kind of schemas that are mapped, so in some cases other function groups could be more frequently used.

2.3.7 Constants

In mathematics, constants are function which have no parameters, and therefore return always the same value. In computer science, constants and functions are often regarded separately, and so it is in schema mapping. Besides, functions which have no parameters do not necessarily return always a unique value, e.g., functions like $random()$ or $getCurrentTime()$.

Many functions get values from the source database as input, i.e., they refer to a source node, but sometimes constants are necessary, too. A prime example is the concatenation of first name and last name. To concatenate only first name and last name of a person would end up in returning values like "PeterWhite", "SusanAlington", etc. Therefore, a constant parameter is often needed, which would be in this case a blank.

Constants appear most frequently in mathematic expressions. In a previous example the total price of a book was calculated by $price * (1 + salesTax)$, so in this case 1 is a constant. When converting measure units or currencies, constants appear as well. Many complex formulas also require constants, even though they do not appear in schema mapping very often.

Sometimes constants are not only used together with functions, but also to generate a default value for a target element. This might be necessary when a target schema has an attribute which the source schema does not have. In this case, a constant could point at the target element, initializing it always with a default value.

Whether constants are used frequently or not, depends, as always, on the context. However, if a program offers functions, it should also offer constants.

2.3.8 Filters

Filters are special constructs that express whether a specific data row is to be mapped or not, or to which part of the target schema a data row is to be mapped. Filters therefore serve as selection or switch, and can be seen as a function as well, or rather as a set of functions, because a filter may typically consist of different functions. Figure 2.2 shows a rather simple filter, which already consists of 6 functions and 3 constants.

Most programs do not offer filters in particular; they simply assume that each data row is to be mapped, since data loss is normally undesirable. However, in some cases a complete merging is not aimed after all, or a data row could be mapped to different parts of the target schema, making filters indispensable then.

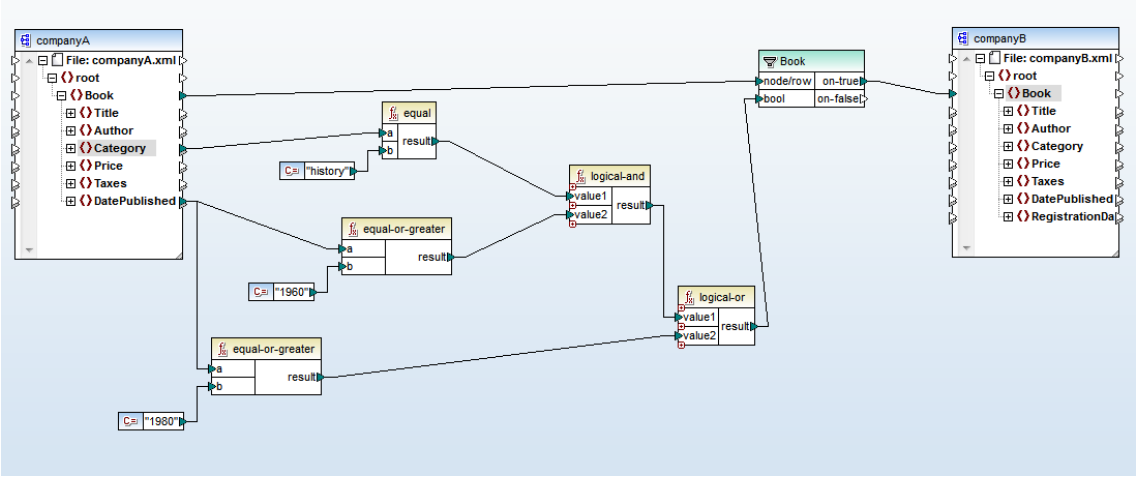


Figure 2.2: A filter created in Altova MapForce 2009. The filter is created out of general functions, and only maps books that were published after 1980 or that were published after 1960 when their genre is history.

An example where filters might be needed is the following: two companies A and B sell books and have rather similar schemas. For instance, a book has an author, a category, a year when it was published, a price and a synopsis. Now both companies decide to incorporate and therefore perform data integration, where the schema of company B will serve as the target schema. They also decide that they won't sell anything but novels anymore. Furthermore, they agree that they will only sell novels which are not in the public domain, that is, which were published after 1922. The filter could be expressed as follows: transform book objects from DB_A to DB_B only if $DB_A.Book.Category = "Novel"$ and $DB_A.Book.Year > 1922$.

This filter can be realized by using functions, i.e., an if-, an equal-, a greater-than- and an and-function. Another way is to apply filters by using a specific language, e.g., $if(and(equal(Book.Category,\ 'Novel'), greater\_than(Book.Year,\ 1922)))$

Filters chiefly consist of relational and logical functions, but also other functions

can be used, just as shown in the following example where only companies that end with "& Co." are mapped: $if(endsWith(Company.Name, '\& Co.'))$.

Filters can not be detected by schema matching algorithms, and at the moment there seems to be no way to detect filters automatically at all, because the possible filters a company could think of are too various and too individual. However, it would be an obvious gain to a schema mapper if at least the possibility of filters exists. Only this way all aspects of data transformations could be covered.

2.4 Conclusion

In this section, the aims, basics and difficulties of data integration were explained, where schema matching and mapping have been the center of interest. Since this chapter is primarily the basis for the next sections, especially the fourth chapter, this conclusion is supposed to summarize the basic concepts again.

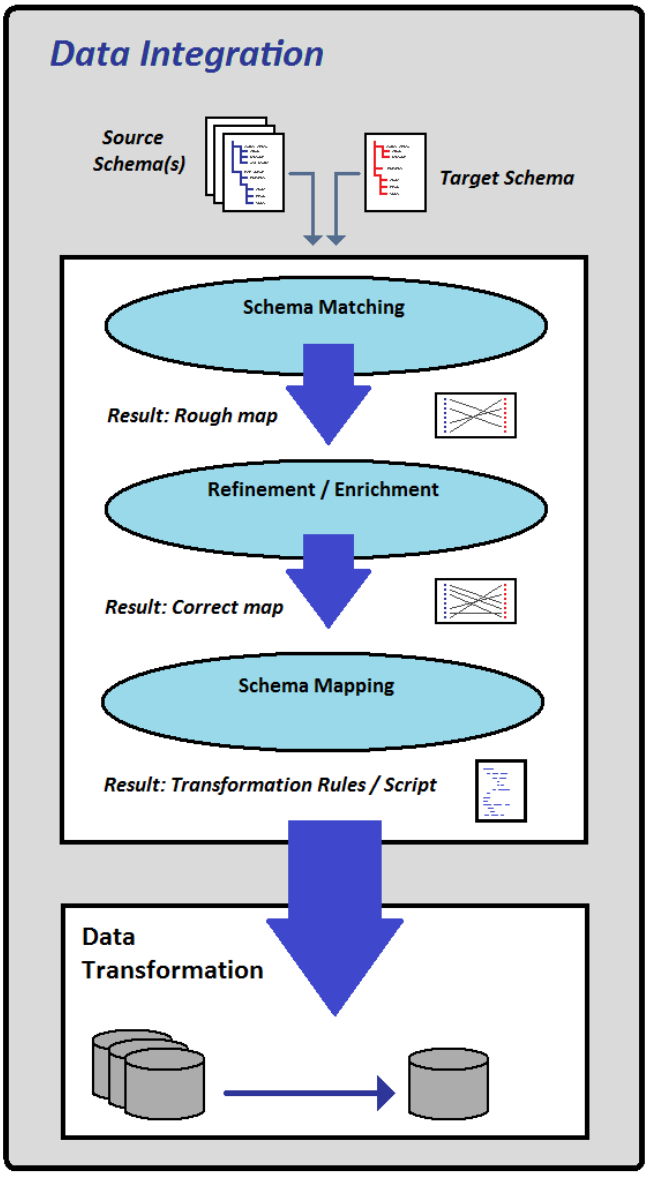


Figure 2.3: The basics of data integration.

Figure 2.3 shows roughly how data integration is usually performed, and therefore might be a good summary for this section. As shown in the picture and explained in this chapter, data integration often consists of three important steps: schema matching, schema mapping and data transformation. A common task is to merge different databases to one database with one specified schema. Eventually, the database must be consistent and, if not desired otherwise, consist of the exact same information the original databases does.

The first step in data integration is normally to perform schema matching, i.e., employing algorithms that detect relationships between the different database schemas. Those relationships, called correspondences, can have different cardinalities, so (1:1), (1:n), (n:1), (n:m), and there are different kind of relationships (equivalence, inclusion, overlap, disjunction). Correspondences whose cardinality is at least (1:n) or (n:1) are called complex correspondences. The result of schema matching is a list of correspondences, called map or alignment. The next step is to refine the map, i.e., add, delete and adjust correspondences. After this, rules are defined in a specific lan-

guage so that the data objects can be transformed at last. This process is called schema mapping. Data transformation is the final step in data integration; it can be performed at one time (physically) or at runtime (virtually).

Since the correspondences alone are not sufficient enough to ensure correct data transformation, functions are needed; they are also needed to allow complex correspondences. A function can have $0..n$ parameters and returns exactly one unique value which is mapped to the target database or to another function. The parameter itself can be a value from the source database, a constant or the return value of another function. There are several function groups, like mathematical functions, string functions and conversion functions. Each function group consists of several functions, yet the importance of these functions differs considerably.

Chapter 3

State of the Art

3.1 Introduction

Before the implementation of functions and complex correspondences will be regarded closer, some existing schema mapping and matching programs shall be introduced and compared with each other. Besides this, some studies and papers that concentrate on complex correspondence detection shall be presented to show what kind of approaches already exist, and what their shortcomings are.

There are basically two classes of schema mapping programs. The first class encompasses commercial applications, which are often established solutions provided by large enterprises like IBM or Microsoft. The second class encompasses research projects or open source projects, which are often less established, but in some cases offer more useful features than the proprietary solutions. However, it is more difficult to obtain those programs, and it seems that many promising approaches and research results are far too often underestimated or neglected.

Many schema mappers already allow complex correspondences, functions and filters, so it might be of great interest to see what kind of functions they offer, how schema mapping is performed, and how they stand out against each other. After having investigated those solutions, it should also become clear how COMA++ was to be improved so that it would be able to keep up with the existing solutions, and even offer features that other solutions does not offer.

There are various requirements with regard to schema mapping programs. Some important requirements are:

- Supporting a variety of input and output formats, e.g., XML schemas, CVS-Files, databases from different database systems, etc.
- Supporting different transformation standards, e.g., XQuery, XSLT, SQL, etc.
- Offering more than one source schema.
- Semi-automatic correspondence detection (and high quality of the match algorithms).
- Little execution times.
- Offering functions to adjust the map, including filters.
- Offering complex correspondences.
- Offering examples and transformation preview (ease of use).
- Appropriate usability and clear user interface, supporting different operating systems.
- Moderate price and appropriate licenses.

Of course, these requirements depend on the respective companies and tasks which are to be performed. However, the requirements presented above might be general requirements most companies would expect from a powerful schema mapper.

In the following, some well-known programs and studies will be shortly presented, where the focus will be primarily on functions and complex correspondences. In chapter 3.2, commercial schema mappers will be presented, and in chapter 3.3 non-commercial solutions, so research and open source projects. After these applications were introduced and assessed, chapter 3.4 focuses on studies and papers which are concerned with complex correspondence detection. The main ideas given in this chapter are finally summarized in chapter 3.5.

3.2 Commercial Schema Mapping Solutions

3.2.1 Microsoft BizTalk Server

An apparently widely known schema mapping solution is Microsoft BizTalk Server, an extensive software for all kind of business processes, which also includes a schema mapper and a data transformator [17]. The latest version is BizTalk Server 2010, to which a free trial is available so that it was possible to examine the program closer.

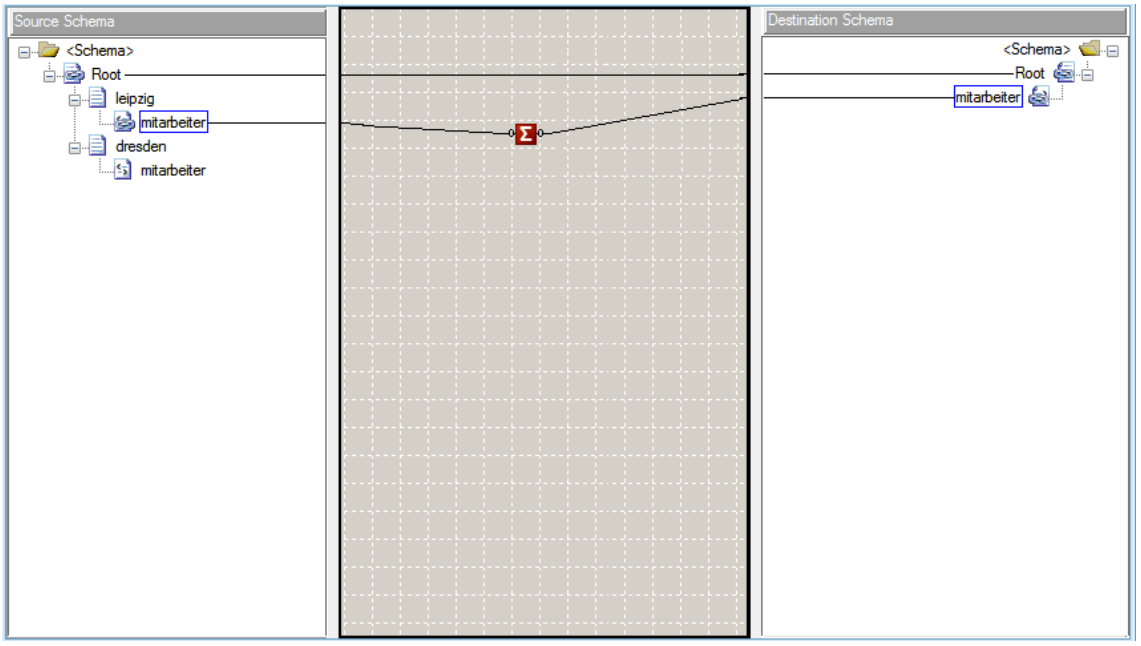


Figure 3.1: A sample map performed in Microsoft BizTalk Server 2010.

As described in [17], Microsoft BizTalk Server is an extensive software for many aspects of data integration, especially focusing on Enterprise Architecture Integration

(EAI), monitoring, diagnosis, Business-to-Business Integration (B2B) and similar fields. One aspect is also schema mapping and data transformation, which is actually just a small part of the entire software.

How powerful Microsoft BizTalk Server might be, the schema mapper offers only very simple (semi-) automatic correspondence detection (mostly string-based detection), and is obviously rather designed for manual schema mapping. However, since BizTalk Server 2009 did not offer any kind of schema matching at all, it is quite possible that Microsoft will considerably improve the schema matchers over the next years.

For now, the schema mapper of BizTalk Server 2010 does not seem quite exceptional, especially since the matching algorithms are only simple ones. Besides, only one source and one target schema may be used, something which ambitious companies might not approve. On the other hand, it turns out that Microsoft BizTalk Server offers a huge amount of functions, this way allowing sophisticated data transformation and complex correspondences. Microsoft BizTalk Server is XML-oriented, i.e., it is only able to deal with XML schemas and XML data; this blurs its features, because a schema mapper should offer much more formats than only xml.

For creating a map, the main window is divided into three parts with the source schema on the left-hand side, an empty panel in the center, and the target schema on the right-hand side. The user can drag lines from source elements to the target schema, this way creating a map.

It is also possible to drag function icons in the central panel (the actual map), and connect it with source and target elements. These icons represent a function. They have ports on the left-hand side, which represent the input parameters, and one port on the right-hand side, which represents the function's output (return value). Microsoft BizTalk Server allows any number of functions between a set of source elements and target elements, thus offering any kind of complex correspondences.

By clicking on the function symbol, a little dialog opens where some basic information about the function is given (e.g., a description of the function and its parameters). In this dialog, constants can be declared as well, but they are not visible in the map then.

From this point of view, Microsoft BizTalk Server 2010 is a rather powerful schema mapper, which offers all kind of possible maps. It also offers functions and expressions to create filters, e.g., the if-function and while-function. One little snag is that these function icons are rather small, and are only distinguished by a unique symbol. In a large map this can soon become confusing for the user.

3.2.2 Stylus Studio

Stylus Studio is an XML-oriented solution which also includes a schema mapping tool [23]. Besides this, Stylus Studio offers various other XML tools, e.g., XPath and XQuery editors, XSLT editors, XML schema and DTD editors, and many other features. Like in Microsoft BizTalk Server, the schema mapper is therefore just a little part of this solution. The full name of the current version is Stylus Studio 2011 XML Enterprise.

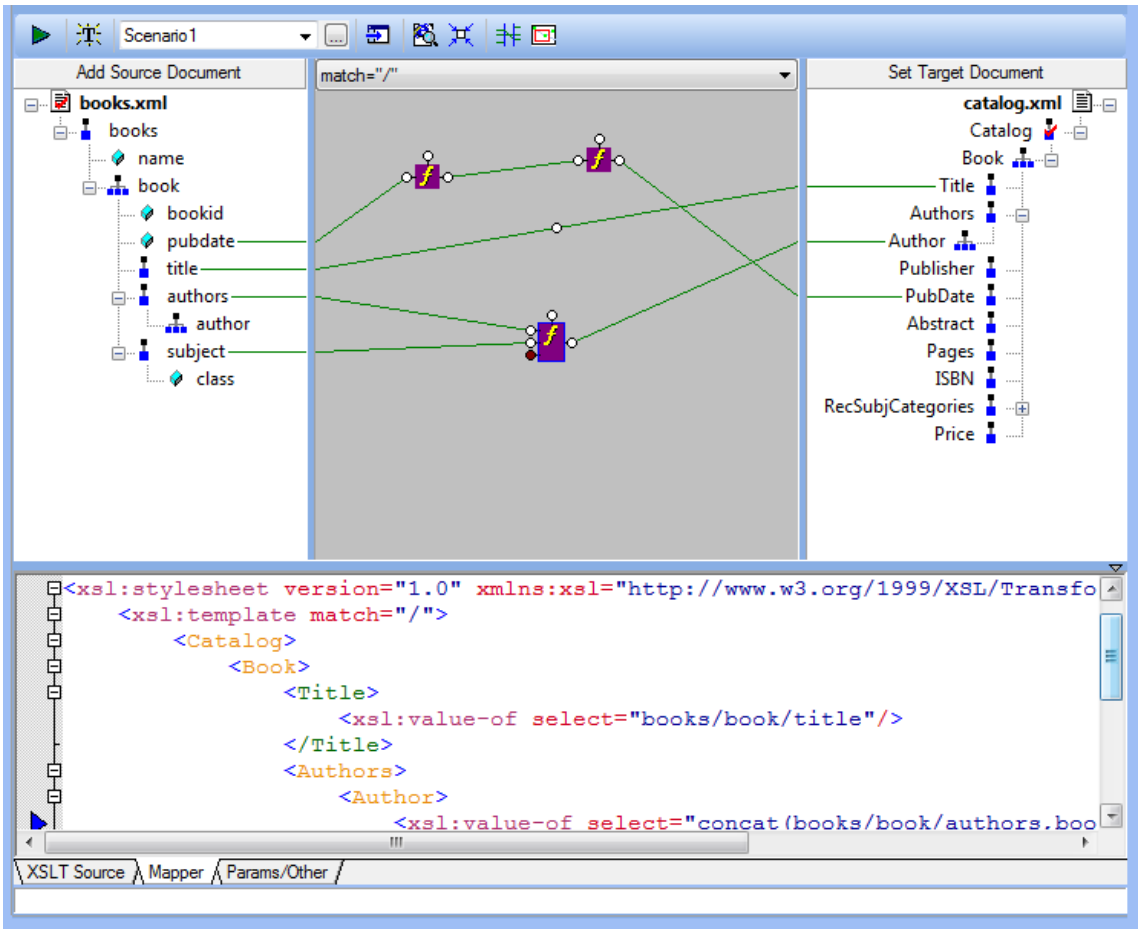


Figure 3.2: A little sample map created in Stylus Studio, as well as the respective XSLT script generated by the program. As it can be seen, the GUI is rather simple, and large mappings can quickly become confusing.

Since Stylus Studio is basically XML-oriented, only XML, XSD, DTD and WSDL formats are supported as source and target schema. For the data transformation, an XSLT script or XQuery script is generated. The user has to decide first which format is to be used; only then the mapping can be created.

The mapping area appears similar to Microsoft BizTalk Server, but is considerably less extensive. Apart from only focusing on XML schemas and XML-based data transformation, Stylus Studio offers less functions. In fact, it comprises just the

functions XSLT 2.0 offers. These include XSLT instructions like *value-of*, *for-each*, *if*, *choose*, *apply-template* and *call-template*, as well as XSLT functions like *floor*, *ceiling*, *concat*, *contains*, etc. If XQuery was selected as transformation standard, functions cannot be used at all.

For most mappings, those XSLT functions should be sufficient, though, and just like in Microsoft BizTalk Server, several functions can be used in a correspondence. Besides this, the XSLT/XQuery script is generated in real-time while the user designs the map. The user is also able to interfere with the script, which is pretty important, since XSLT does not provide arithmetic functions like *add* or *multiply* so that the user is obliged to add such expressions to the XSLT script manually. Stylus Studio does not offer schema matching, and therefore is just a simple schema mapper.

Just as in Microsoft BizTalk Server, the functions are depicted as little icons with ports on the left-hand and right-hand side for the input parameters and the return value. Worse than in Microsoft BizTalk Server, each function has the same icon symbol so that the user can only distinguish them by placing the mouse over the icon.

Stylus Studio enables to declare constants and relational operations like *equal*, *greater-than*, etc. Since XSLT instructions can be used as well, the software also enables filters. Furthermore, it offers the possibility of using functions from external Java classes, which can be linked to the program. This way, the small function set (about 25 functions) can be enhanced.

It should have been already pointed out that Stylus Studio is not a very powerful schema mapper. It only focuses on XML schemas, it only creates mappings in XSLT or XQuery, and the number of functions and features is rather small. The mapper of Stylus Studios appears to be an XSLT/XQuery editor after all (and for this purpose it is quite good), but for real schema mapping does not seem recommendable. However, there exists another proprietary solution which is much more promising. This is Altova MapForce 2011 and will be investigated next.

3.2.3 Altova MapForce 2011

While Microsoft BizTalk Server and Stylus Studio offer a schema mapping tool among many other features, Altova MapForce is an application which is only designed for schema mapping [3]. It therefore is one of the best schema mapping programs, even though it does not contain many schema matching algorithms either.

Unlike the solutions presented before, Altova MapForce offers various input and output formats, something a sophisticated schema mapper should offer by all means. It allows mappings between XML files and databases, flat files (csv), EDI, Excel

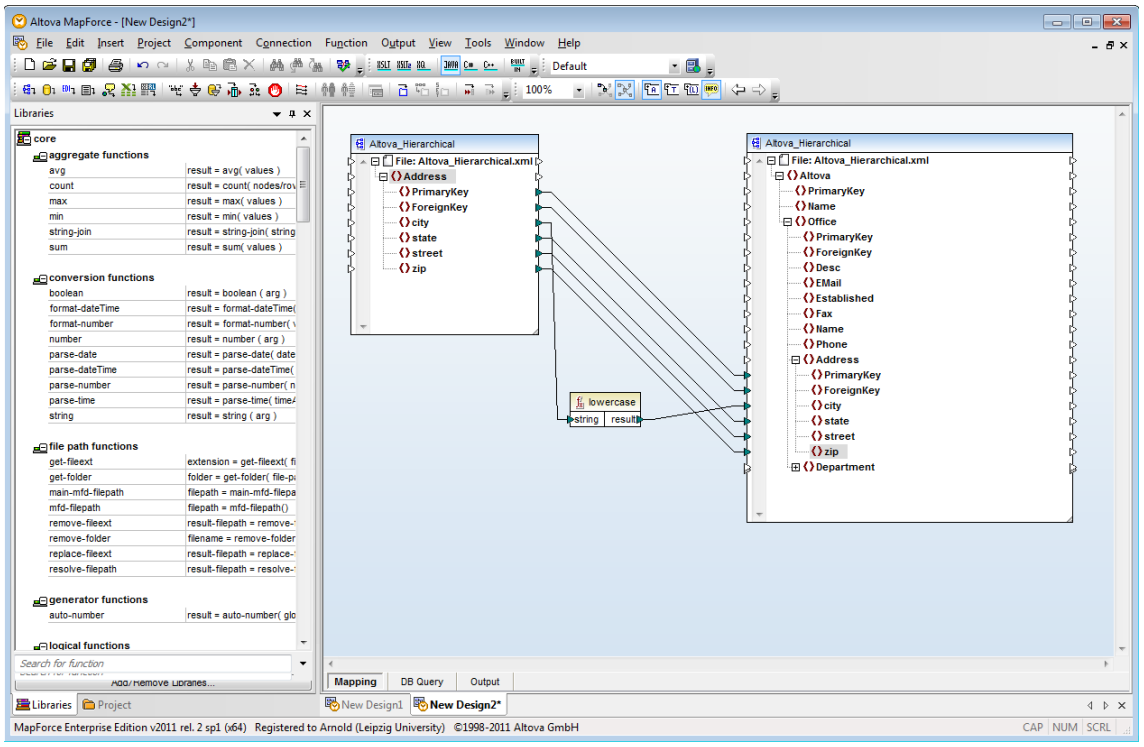


Figure 3.3: Screenshot of Altova MapForce 2010 version.

2007 and other formats. The mapping itself can be encoded in XSLT or XQuery, but Altova MapForce also supplies a code generator which enables the user to convert the map into a C++ or Java program. Altova MapForce can also deal with more than one source schema and allows chained transformation. Moreover, the application allows the user to connect source instance files with the map so that a preview of the data transformation can be displayed.

The program excels especially due to a clear GUI and ease of use. Functions are represented as boxes, and are connected with source elements, target elements, constants or other functions. In each function box the function's name is displayed so that the map remains clear, even if many functions are used. Like functions, constants are represented as a small box; the value is always visible.

While there had no matching algorithms been implemented in the 2009 version, a few techniques were added to the 2010 version. This includes a name matcher and a structural matcher. However, they are practically the simplest schema matchers possible, and it is doubtful whether the term "matcher" even fits in this case. The name matcher connects elements which have the very exact name, and the structural matcher draws lines from each source element to the very opposite target element. Hence, there are no intelligent algorithms working in the background, and for most schemas the matcher will be of no use. However, since some basic matchers were implemented in the 2010 version, it could turn out that these matching algorithms will be improved in the future. From this point of view, Altova MapForce and Microsoft BizTalk Server could become very competitive.

Altova MapForce offers a large amount of functions, even much more than Microsoft BizTalk Server does. There are about 200 different functions, which also include conditions and relational expressions so that filters can be created. Besides this, user-defined functions are supported.

With respect to the many great features Altova MapForce offers, it appears to be a sophisticated schema mapper which allows extensive and ambitious schema mapping. It would be even an outstanding solution if it offers enhanced match algorithms.

3.2.4 Further Applications

The applications presented above are only a selection of the programs which were examined in the context of this Master's thesis. Some further approaches and projects are itemized in [2], however, to keep this study at a moderate length and to concentrate on more profound subjects, they shall not be explained more precisely.

The IBM Infosphere Data Architect [10] is a powerful schema mapper and schema matcher, which is oriented toward database mappings and allows different database systems (MySQL, Oracle, DB2, etc.). It also offers many functions and allows complex correspondences. The Oracle Data Service Integrator [19] is another solution for database mapping, but focuses strongly on Oracle databases, and does not work smoothly on Windows operating systems.

3.3 Research Projects

3.3.1 OpenII

OpenII stands for Open Information Integration, and is an open source schema matching and mapping program, which offers mappings between different kind of schemas [18] [21]. It seems to be a rather promising program, because it offers sophisticated schema mapping, different functions and a rather good layout. Lately there is few activity on the code, though; the latest version available is from October 4, 2010, more than half a year back, and temporarily there does not seem to be many commits either (as of April 2011).

OpenII is run in the common Eclipse environment. It is rather easy to use, and offers multiple input and output formats, such as XML, relational, Domain, RMap, GSIP and others. Similar to COMA++, the workflow consists of different matching

algorithms: name similarity matcher, documentation matcher, mapping matcher, exact matcher and word matcher. However, the workflow is considerably less complex than in COMA++, and the only settings the user may adjust, is to enable or disable a matcher.

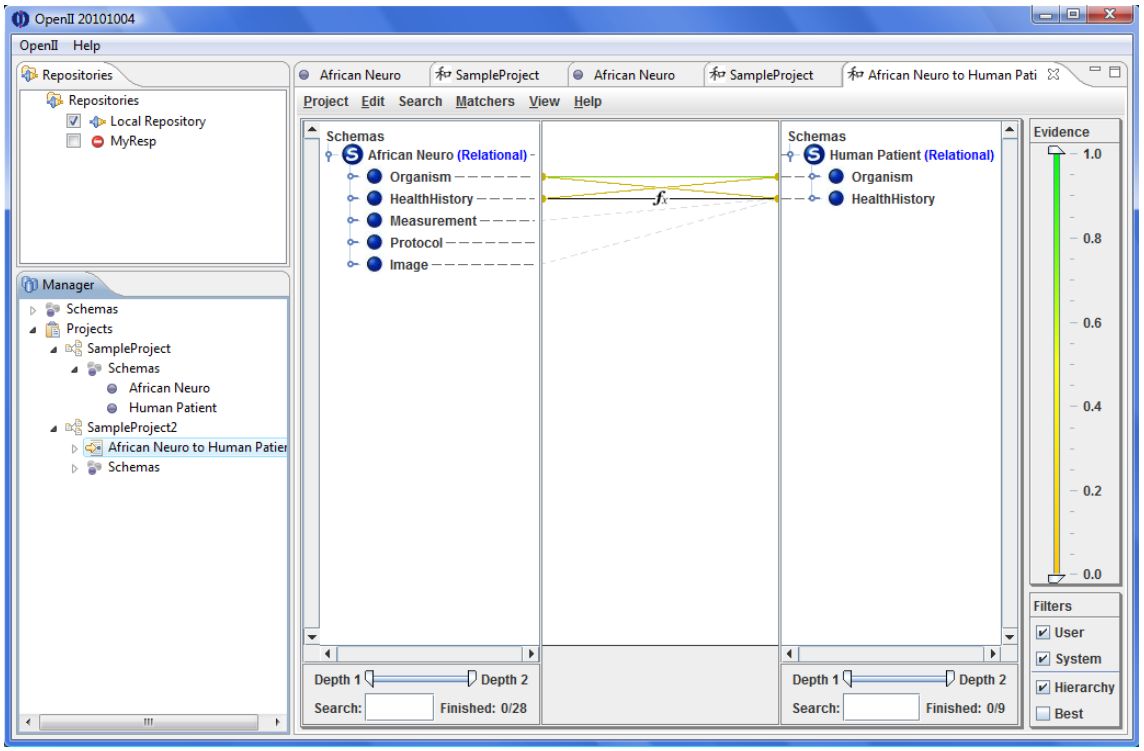


Figure 3.4: OpenII is a rather promising program, because it offers several schema matching algorithms, functions, a good layout, and is an open source project. Whether it will ever be completed, is not sure, though.

The layout seems pretty nice, especially since OpenII presents the matches in different colors according to their confidence. The user can also alter the confidence domain to display less correspondences (e.g., a domain from 100 to 75 would only present the matches having a confidence above 0.75).

OpenII provides a couple of functions, however, only one function per correspondence can be assigned. There are only some basic functions implemented so far, e.g., *lower*, *average* and *ceiling*. To declare a function, the user opens a new dialog, selects a function from a combo box and adds the parameters from another combo box. The function will be displayed by an f_x symbol on the correspondence line.

Thus, the way how functions are implemented in OpenII is very simple and cannot stick with programs like Microsoft BizTalk Server, not to mention Altova MapForce 2011. Also, even the most important functions like *concatenation*, *split* and the arithmetic functions are completely missing, and declaring functions is more complicated than in the other programs. The lack of the aforementioned functions also entails that no complex correspondences can be created.

OpenII seems to be an interesting approach, and has certainly much potential, but it is not sure whether it will really come to a successful end. There still seem to be some errors in the program, and regarding the current version, it seems not possible to produce all possible kind of maps.

3.3.2 Clio

Clio is another well-known schema mapping management program, which is now a research project of IBM [11]. It offers matching, mapping and data transformation solutions, and allows mappings between any combination of XML and relational schemas. Besides regular matching algorithms, Clio scores since it discovers and regards data constraints [14].

At the moment, it seems as if no current version of Clio can be obtained. Whether it will be extended and be a powerful schema mapper someday, is rather uncertain.

3.3.3 Spicy

Spicy is another promising schema mapping tool [28]. Developed at the University of Basilicata, Italy, it especially excels on account of its clear GUI and ease of use. Spicy can deal both with relational Schemas and XML schemas, and also exploits instance data to perform a sophisticated mapping.

The latest version of Spicy also offers functions and constants, but the functions have to be created manually, and there is no function list containing the available functions yet. However, Spicy checks the syntax and displays an error message if it is not correct. Since Spicy is able to use functions, all kind of complex correspondences can be expressed. Besides this, Spicy offers filters and direct data transformation [16], thus making it a complete and valuable schema mapping solution.

3.3.4 Further Applications

Heptox [7] and Falcon [12] are former research projects which got apparently abandoned. RiMoM [24] is a research project which focuses on ontology matching, however, it seems as if this project has been abandoned as well.

3.4 Studies

3.4.1 The iMAP Dissertation

Introduction

A study which is largely dedicated to complex matches is the dissertation of the iMAP project, a schema matcher which especially concentrates on complex match discovery. It seems to be a rather promising project with regard to complex correspondence detection, but unfortunately the iMAP application is not available yet, and it is very uncertain whether the project is still pursued.

Basics

iMAP [15] consists of several searchers where each searcher regards a specific subject like string expressions or numerical expressions. The seachers therefore only regard a special part of all possible correspondences, mainly based on the attribute types, combination operators and heuristics. Thus, iMAP tries to prevent pointless correspondences like $(BookPrice,\ multiply(Title,\ Year))$.

One important feature of iMAP is that it especially regards instance data of databases to substantiate or refute a supposed correspondence. For instance, iMAP counts the number of unique values connected with a database element, and after this assumes that the matching elements in the target database should have a similar number of distinctive values. iMAP also tries to draw conclusions out of duplicates, that is, if source and target database share any duplicates.

For running the several searchers, a search technique called *beam search* is used in iMAP. The basic idea is to use a scoring function which calculates the confidence between two sets of nodes. Just like all other schema matchers, it starts with detecting (1:1)-correspondences by comparing each node with each other and calculating the score. After this is done, the matches which does not exceed a specified threshold are removed, and the second iteration begins. Now each of the remaining elements are compared with the combinations of two remaining elements, and the score for this match is calculated (note that this is a complex match now). Then all correspondences which does not reach the threshold are removed again, and the third iteration commences. When no further correspondences can be found (so no further correspondence reaches the threshold), the algorithm terminates.

The score is calculated by several methods like using instance data, background knowledge and machine learning. Since the developers of iMAP place great value on modularity, each searcher uses its own measure methods. The program also

enables user feedback and explains the score of correspondences (e.g., why some correspondences were scored higher and others lower) [15].

Techniques

In the following, the several searchers of iMAP will be shortly introduced, including the strategies they use to detect complex matches.

The *text searcher* of iMAP concentrates on beam search, and this way discovers complex string correspondences. To score the element combinations, machine learning, instance data and background knowledge is used, yet no further explanations about the details of detecting complex string correspondences are given in the paper.

The *numeric searcher* tries to find complex matches by comparing its distribution. It is also limited to some basic operations like arithmetic ones to reduce the number of comparisons. The distribution of values is calculated on the basis of the Kullback-Leibler divergence measure.

The *schema mismatch searcher* detects matches between instance data and element names, so in this module instances and elements (which are normally completely different things) coalesce. The developers of iMAP justify this searcher by stating that in many cases the element name appears (or at least partly appears) in the instances. This way a link between the elements laptop and notebook could be detected if values like "travelers notebook", "long battery life notebook", etc. appear in the laptop element.

The *category searcher* first determines the elements which probably represent a category. A category is considered an element which has only a few distinct values, the threshold specified in the paper was 10. If all categories are determined, they are compared with each other. If two categories have the same number of distinct values (or almost the same number) they are considered a match. In the next step, a function is built which tries to assign each value from the source instance to the correct value of the target instance, basically on the basis of a distribution function again.

The *unit searcher* discovers conversion functions between elements which express the same thing using different measure units (like centimeters and inches). The algorithm therefore searches for a special unit token (like in, cm, etc.), both in the element name and instance data. It then creates the conversion function on the basis of commonly used measure units.

The *date searcher* detects (complex) correspondences between dates, e.g., (*date*, *concat*(*day*, *month*, *year*)). It is based on a date ontology to discover those date correspondences, however, nothing about the many different date formats is mentioned in this study [15].

Evaluation

The developers of iMAP tested their solution on four different real-world domains, and calculated recall and precision both for (1:1)-matches and complex matches. For this, each target element was regarded, and it was checked whether the correct source element was assigned or not.

The accuracy for (1:1)-matches was, depending on the domain and the used strategy, between 62 and 92 %. Considering the top-3 accuracy (meaning that the correct correspondence was not necessarily found, but was among the 3 correspondences iMAP deemed as most likely), the accuracy was between 64 and 95 %.

As one would generally expect, the accuracy for complex matches was considerably smaller. The top-1 accuracy over all strategies was 33 .. 55 %, and the top-3 accuracy 43 .. 92 %. However, if one regards only partly discovered correspondences (so the correspondence was detected in general, but some elements were missing or did not belong to it), the accuracy is much higher, 36 .. 86 % for the top-1 accuracy, and even 70 .. 100 % for the top-3 accuracy [15].

Problems in Finding Complex Correspondence

The developers of iMAP give the following reasons why the accuracy of complex correspondences is rather small in comparison to the (1:1)-matches [15]:

- It frequently occurs that iMAP detects the correct correspondence, but that some elements are missing.
- On the contrary, it might happen that too many elements are assigned to a correspondence, like $concat(title,\ firstname,\ lastname,\ companyname)$.
- If the databases are disjoint, it is difficult to detect the correct numerical expression. In the evolution part of the study it can be seen that the accuracy is always smaller if disjoint databases are regarded instead of overlapping ones.

3.4.2 STBenchmark

STBenchmark [2] [29] does not give any solutions for complex correspondences and function detection, but is a benchmark to test the quality of schema mappers. For this, STBenchmark offers 17 scenarios, each containing a source and target schema and a few correspondences. A perfect schema mapper would now create the correct

transformation scripts for all scenarios. Some scenarios are pretty simple and can be solved by all schema mappers (e.g., the correspondence between address and address), while others are more complicated and cannot be solved by most of the schema mappers.

In the context of this thesis, STBenchmark was used to specify which matching tasks were to be solved, although STBenchmark does not focus on schema matching, but on schema mapping. The scenario which was most interesting for this study was scenario 13, which is called *Atomic Value Management*. It shows two complex correspondences ($name$, ($firstname$, $lastname$)) and (($street$, $city$, zip), $address$) which a schema matcher must detect to fulfill this scenario.

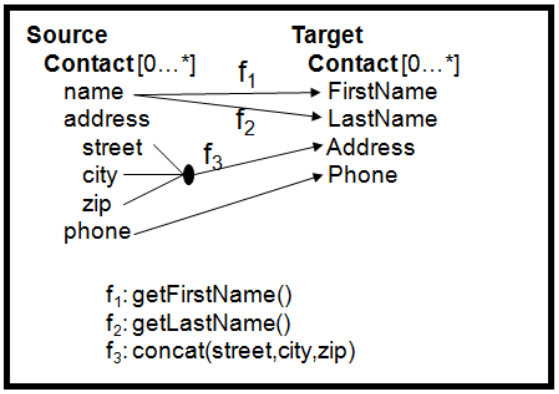


Figure 3.5: Scenario 13 of the ST Benchmark Scenarios, http://www.stbenchmark.org/basic.htm#atomic. It was one of our main goals to develop strategies which fulfill this scenario and similar ones.

STBenchmark also expects the mappers to apply the correct connector functions to the correspondence, something a reliable schema matcher and mapper should generally offer. For the first correspondence, the two functions $getFirstName(name)$ and $getLastName(name)$ have to be detected, and for the second correspondence the function $concat(street, city, zip)$. However, the functions $getFirstName()$ and $getLastName()$ are rather specific functions, which even established schema mappers do not offer – using the split function and splitting by the blank would be a much easier (but also less reliable) method. On the contrary, the concatenation function for the second correspondence seems too simple. To create correct address strings, separators (normally blanks) must be inserted between street and city as well as city and zip. Regarding these two aspects, it is not clear how much value STBenchmarks really places on generating the correct functions.

It can be assumed that currently no schema matcher can solve scenario 13, except for the iMAP approach, but which is not available yet. It was one goal of this study to implement strategies to detect complex correspondences, the respective functions, as well as general functions, so to solve scenario 13 was one important task as well.

3.4.3 Further Studies

Another approach about complex correspondence detection was described in [33], which does not focus on schema matching, but on graph comparison and graph matching. The study could appear pretty promising, because it gives solutions to detect many-to-many correspondences between graphs, and schemas are typically represented as graphs after all, however, the solution presented in the paper tends to a discrete optimization problem, and seems rather inconvenient for schema matching. It does not consider any semantic aspects.

In [5] they rather give an overview about schema matching than some specific solutions, but in one section also concentrate on complex matches [5]. However, it is rather a short itemization of approaches in complex match discovery, and seems to have a strong relation to the iMAP study. Some results from this investigation are that domain knowledge and domain ontologies seem inevitable to perform complex correspondence detection, just as performed in iMAP.

The study about web query interfaces [9] is also dedicated to complex matches, and focuses on matching web interfaces which belong to the same domain. For this, a framework was developed that offers a complex matching algorithm, the DCM (domain correlation mining), mainly based on linguistic relations. The main idea is that complex correspondences can be detected by analyzing co-occurrences and synonyms, while co-occurrences suggest a complex correspondence and synonyms do not. Thus, correlation mining would be useful to detect complex correspondences. The paper therefore rather concentrates on linguistic aspects, and like [33] draws on mathematical approaches.

3.4.4 Assessment

Analyzing the approaches revealed that while at least a few studies concentrate on the wide problem of complex match detection, not a single one regards function detection. To give an approach in automatic function detection was therefore an important goal of this thesis.

After all, it seems that iMAP is the only established study about detecting complex correspondences in schema matching. Although there are a couple of other papers available, some mentioned in the previous section, they mostly do not concentrate on complex correspondence detection per se; they either formulate the problem (but do not give a solution), or give rather specified approaches which are hardly useful for

classic schema matching (although some approaches might be adapted for schema matching anyway).

For the solution which was developed at the WDI Lab, we therefore rather developed strategies ourselves, and did not draw on the ideas presented in those papers. We especially developed a solution which does not necessarily rely on background knowledge, domain knowledge or machine learning, but rather exploits the schema structure and simple correspondences of a map to detect complex matches. We also developed strategies to discover general functions between correspondences, which mainly operate on instance data.

3.5 Conclusion

The applications and studies presented in this section are only a selection of the currently most famous or most-cited solutions, but it should have been pointed out that the programs considerably differ from each other.

As mentioned before, each program can be roughly assigned to one of the two groups (commercial solution or research project). The investigation's result was that the commercial solutions enable nearly all important aspects of schema mapping, especially supporting complex correspondences, functions and filters, but hardly concentrate on schema matching. On the other hand, the research projects rather concentrate on schema matching, offering astonishing matching algorithms, but hardly offer complex correspondences and functions, which makes it impossible to build complete maps. A program which is really good in both aspects could not be found yet.

Now a schema mapping program which combines the benefits of the first and the second class, so the clear layout, functions, filters and multiple file formats (like in Altova MapForce), as well as the established matching strategies (like in OpenII or the solution presented in the iMAP dissertation), could prove to be a real gain in the large list of already existing solutions. For this, COMA++, which is a classic schema matcher and belongs to the second group, was extended by features to keep up with the commercial solutions.

Some further feature that neither the commercial solutions nor the research projects offer, is to detect functions and complex correspondences automatically. This was something which was implemented in COMA++ as well so that it does not only combine the advantages of both groups of schema mapping solutions, but also offers a completely new feature.

Chapter 4

Function and Complex Correspondence Detection

4.1 Introduction

4.1.1 Preface

After the basics of functions and complex correspondences were described, and approaches and applications were presented, this chapter will eventually describe how complex correspondences and functions were implemented in COMA++, and which strategies were developed to discover them automatically. Now concentrating on the implementation, so on the practical part, this chapter will make up roughly half of the Master's thesis, and contains the actual results of our work. It will start with some general information about COMA++ and the tasks which had to be performed, and after this will concentrate on the implementation of complex correspondences and functions in COMA++, as well as the strategies invented for function and complex correspondence detection. At the end, an evaluation will be given, which is mostly based on test cases we carried out. This evaluation will show how the program behaves in practice and which benefits and drawbacks result.

4.1.2 The COMA++ Schema Matcher

COMA++ is a classic schema matching program that compares two database schemas by exploiting different matching strategies, and subsequently returns a list of (1:1)-correspondences (map) between those schemas. From that point of view, it basically works like many other schema matchers, although it excels due to a large amount of different matching algorithms and matching strategies that have been developed at the database chair over many years.

Like other schema matchers, the classic COMA++ did not offer complex correspondences or functions. It was therefore decided to enhance the software by complex correspondences and functions, just as they are offered in many schema mappers like Altova MapForce, Microsoft BizTalk Server, Stylus Studio, etc. This would make COMA++ a full-fledged schema matcher, as well as a schema mapper offering functions and many-to-many-correspondences, something that apparently does not exist yet. However, it still seemed that this innovation did not go far enough, because for this goal no new strategies had to be implemented, and it rather would have been combining the two pretty things that already exist, just in different applications. In point of fact, adding functions and the possibility of complex correspondences to COMA++ could have been performed within a few weeks. It was therefore decided to offer an approach in complex correspondence and function detection as well, something which no other program can offer at the moment (except for the iMAP project, but which is not available yet), and something which is also much more intricate than just offering functions for manual schema mapping.

With this, we had two main goals: first we wanted to extend COMA++ by func-

tions and complex correspondences so that all kind of mappings can be expressed and direct data transformation may be performed. Second, we wanted to implement and present an approach about complex correspondence detection and function detection.

4.1.3 Background and Statistics

Implementing functions and complex correspondences was performed at the WDI Lab, where the new COMA++ version was developed. The implementation followed after a precise competitive analysis of schema mappers and schema matchers, as well as a thorough requirement analysis. At this point, we decided which functions were to be offered, and in which way the detection of them could be realized.

The implementation of the aims presented above took place in a separate project, called coma-mapping. Although being a subproject of COMA++, it was completely independent from the actual COMA++ software so that the implementation did not impair the teams working on other COMA++ sub-projects. The logical integration of the project took place when it was finished, and due to accurate design, went rather smoothly.

Since COMA++ is written in Java, the coma-mapping module was written in Java as well. A survey of the program code developed in the context of this Master's thesis revealed that there are 77 Java classes with altogether 15,765 lines of code. More specifically, there are 7,477 actual lines of code (ca. 47 %) and 3,522 comment lines (ca. 22 %). The rest (ca. 31 %) are blank lines. Every global variable, every method, every class and every package is described in JavaDoc, and all basic criteria for Java code were satisfied.

The 77 classes are located in about 30 packages and have an overall size of 444 KB. We assume that about 65 % of the code is used when the program is run in standard mode; the remaining 35 % are classes for debugging or to run test cases (ca. 25 %), as well as abandonded code (strategies we developed, but did not include in the final version for now; ca. 10 %).

4.1.4 Tasks and Aims

Our specific aims in the context of this Master's thesis were:

- Offer complex correspondences.
- Offer a set of most important functions, altogether about 10 .. 20 functions.
- Offer strategies to detect complex correspondences.
- Offer strategies to detect functions.
- Offer direct data transformation.

Many other tasks came along with these aims. The data structure of COMA++ had to be adjusted to express complex correspondences, the project had to be integrated into the superior COMA++ project, and the GUI had to be changed in many ways. However, the GUI tier was not part of this study and will not be regarded here.

Documentation and tests were further crucial steps. For the documentation, a 50 pages program description was written in parallel to this Master's thesis. Unlike this thesis, it rather describes the structure of the project and how the algorithms are implemented, mainly for the sake of maintenance, whereas this study rather describes the general ideas, concepts and principles. We also implemented about 100 test cases to verify the program. They are all described in the program description as well, so we took tests and documentation very serious.

For the first version of the function and complex correspondence detector, we decided to allow only one function per correspondence, however, we devloped the data structure in a way that it can be easily extended so that the usage of more than one function within a correspondence can be realized in a later version of COMA++.

4.2 The New Data Structure for COMA++

4.2.1 Introduction

The original COMA++ only detects (1:1)-correspondences, that is, it compares each element with each other, computes the confidence, and if it is greater than 0 (or greater than a specified threshold), considers it a correspondence. The correspondences are represented in a matrix, the *coma match result matrix*. As it can be seen in the picture below, each source element is compared with all target elements and vice versa. The combinations that have a confidence above 0 are correspondences.

Representing correspondences between source and target schema in such a matrix works perfectly for (1:1)-correspondences, but there are several reasons why it does not work for complex correspondences:

- A complex correspondence has more than one source node or more than one target node, and each node can take part in different correspondences at the same time.
- Each correspondence may have a function, which has to be expressed.
- Each function may have several parameters. The parameters can be both node references and constants.

	Person_ID	Name	Telephone	Address
PID	**0.2**	0.0	0.0	0.0
First Name	0.0	**0.6**	0.0	0.0
Last Name	0.0	**0.6**	0.0	0.0
Address	0.0	0.0	0.0	**1.0**
Email	0.0	0.0	0.0	0.0
Tel_Number	0.0	0.0	**0.1**	0.0

Correspondences:
(PID, Person_ID)
(First Name, Name)
(Last Name, Name)
(Address, Address)
(Tel_Number, Telephone)

Figure 4.1: Little example showing how the coma match result matrix could look like, and what correspondences it expresses.

For these requirements the coma match result matrix was too simple so that a much more complex data structure was needed. As it can be seen in Figure 4.1, the complex correspondence $(name, (firstname, lastname))$ cannot be correctly expressed, but is represented as two simple correspondences. Our first step was therefore to create a data structure which allows to represent all kind of correspondences and all kind of functions.

4.2.2 The New Data Structure

The new data structure is able to deal with all kind of correspondences, so both complex correspondences and classic (1:1)-correspondences. The picture below shows the UML class diagram of this structure which will now be explained.

Map

The *map* is the top-level class in the data structure, having a source and a target schema, a source and a target instance, and an alignment. Of course, the source and target schema must be known, whereas the source and target instance can be null, i.e., they are optional. The alignment represents a set of correspondences and is the heart of the map. At the beginning, the alignment is equivalent to the match result obtained from the COMA++ matcher, so it is a list of (1:1)-correspondences (the coma match result). Later, the program will enhance these simple correspondences

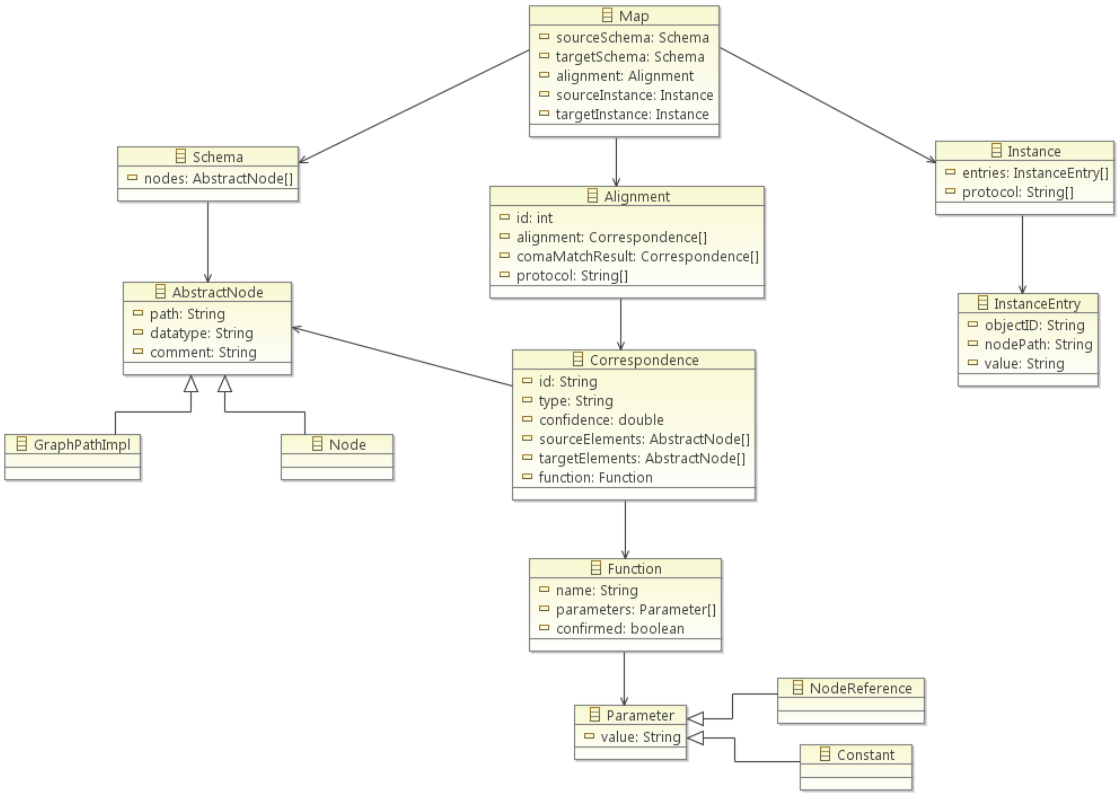


Figure 4.2: UML class diagram of the new data structure to express complex correspondences (simplified).

and add new ones to the alignment, so the function and complex correspondence detector will mainly operate on this coma match result.

Alignment

An *alignment* is a list of correspondences, but this class also stores the type of the source schema, the type of the target schema, a protocol, and especially the match result matrix. The match result matrix is the base for the complex correspondence detection and function detection, but will only be read, so never gets changed.

Before the program can be run, the alignment has to be initialized by using the coma match result. At the beginning, the match result list and alignment are therefore semantically identical, but detected complex correspondences and functions will only be added to the alignment list. The alignment list is therefore the result after the detection process is over, and can be seen as an enhanced COMA++ match result. In the worst case (i.e., no complex correspondences and no functions were detected) it is semantically equal to the coma match result matrix. Otherwise it is (hopefully) semantically richer. If the user adds complex correspondences and functions manually to the map, these changes are also updated in the alignment, but not in the coma match result matrix.

The protocol is a list of protocol entries which are generated when the program detects new correspondences. Also general notes will be added to this list so that the user is able to see what correspondences and functions have been created or changed, and for what reason several possible correspondences or functions have not been created (e.g., because a specific threshold has not been reached etc.).

Correspondence

A *correspondence* has a list of source nodes, a list of target nodes and a function in between. It also has an id, a type (equal, is-a, inverse-is-a) and a confidence value. The confidence value is not used yet. The feature of having a list of source nodes and a list of target nodes is the centerpiece of the data structure – at this point complex correspondences become possible.

The original COMA++ correspondences produced by the COMA++ matchers are therefore data objects which have exactly one source and one target node, an automatically generated id, and the confidence value from the match result matrix. The function will be null (if the function is null, the program considers the function as the identity function, so it simply maps the source value to the target element).

Function

A *function* has a unique name, a list of parameters, and a boolean value "confirmed". The latter specifies whether an automatically generated function was confirmed by the user or not. By default, automatically detected functions are not confirmed, indicating that it is not certain whether this function is correct or desired by the user.

A *parameter* is either a *node reference* or a *constant*. If it is a node reference, the instance value of this node will be used as input when it comes to data transformation. If it is a constant, only this value will be used. For example, in the function $concat(FirstName, \text{ } " \text{ } ", LastName)$ $FirstName$ and $LastName$ are node references whereas " " is a constant.

Instance

An *instance* is a list of instance entries, which also has a protocol used for data transformation logging. An *instance entry* consists of an object id, a node path and a value. The object id describes what object is regarded, e.g., Student874. The node path specifies which node is regarded (e.g., name.firstname), and the value specifies the actual value (e.g., "John"). Having these 3 pieces of information, there exist several methods which return all values of a node (e.g., return all first names in

the database), or all values of a specified object (e.g., return all information about Student 874). These methods help a lot in instance-based function detection.

In a typical instance there are many data objects (e.g., 25,000 students), and each object has a specified number of (node, value) pairs (the number of nodes is specified in the schemas, of course). As matter of fact, some values can be null.

Extension

If the program is to be extended so that it can offer more than one function in a correspondence, a third function parameter has to be created: a function reference. This is because a function could get the return value of another function as its input then. In this case, the Correspondence class must offer a list of functions instead of a single function object. Also, each function must have a unique id so that a function parameter is able to point at it.

Loading and Writing

Besides offering a new data structure, it was also necessary to offer methods to write and load mappings. This was not only needed for test cases, but also to make sure that the user is able to load and save mappings. Each mapping is therefore stored in an XML file, which can be seen as a list of correspondence entries, where a correspondence might look as shown in the picture below.

The XML file has almost the same structure as the internal data structure explained above. Since all information of a map object have to be saved and be loaded again, there was no reason to design the XML specification much different from the data structure. For loading and writing mappings we use a SAX parser, which is able to load and write even large XML documents in a very short time.

4.3 The Functions

It was agreed that the program must offer functions a user can manually apply to a correspondence, as well as complex correspondence and function detection as far as automatic detection can be realized. For this, we had to implement the functions first. Amongst other things, a function has a name, a category, a description, an example, a list of allowed parameters, etc. A parameter again has an allowed type or

```
<correspondence>
    <cor_id>1</cor_id>
    <type>equal</type>
    <confidence>1.0</confidence>
    <src>
        <node>Name.Title</node>
        <node>Name.FirstName</node>
        <node>Name.LastName</node>
    </src>
    <tar>
        <node>Name</node>
    </tar>
    <function>
        <func_name>concat</func_name>
        <parameters>
            <nodereference>Name.Title</nodereference>
            <constant> </constant>
            <nodereference>Name.FirstName</nodereference>
            <constant> </constant>
            <nodereference>Name.LastName</nodereference>
        </parameters>
    </function>
</correspondence>
```

Listing 4.1: Representation of a correspondence in the alignment file.

a list of allowed types. We considered description and examples especially important, because the competition analysis had revealed that some programs do not care much about these information, hence making it hard for the user to understand the meaning and correct usage of functions.

The functions are stored in an XML file, and will be loaded whenever the program is started or called by another program (e.g., the COMA++ matchers). A function in the XML file looks like depicted below.

One rather interesting parameter is the priority value. Since in the first version of the program only one function can be applied to a correspondence, but the function detector could possibly detect more than one necessary function, it has to be decided which function will be used after all. For this, each function has a unique priority, and in case that more than one function was detected, the function having the highest priority will be chosen.

At this point, it should become obvious that we have to deal with two kind of functions in general: the function which is part of a correspondence as explained in the previous section, and the kind of function which is part of the function set COMA++ offers, explained in this section. To avoid misunderstandings, the latter are called *meta-function*, because they simply describe the functions a user might

```
<function>
    <name>lowercase</name>
    <category>String</category>
    <priority>3</priority>
    <description>This function transforms all letters into
       lower case.</description>
    <parameterdescription>This function uses exactly one
       parameter. It is the string which is to be
       transformed into lower case.</parameterdescription>
    <example>lowercase( "New York, USA") -> "new york, usa".
      </example>
    <maxParameters>1</maxParameters>
    <minParameters>1</minParameters>
    <parameters>
        <parameter>
            <from>1</from>
            <to>1</to>
            <type>string</type>
            <mandatory>true</mandatory>
        </parameter>
    </parameters>
    <returnType>string</returnType>
</function>
```

Listing 4.2: Representation of a meta-function.

apply to a correspondence. The functions used in a correspondence can be seen as instances of these meta-functions and will still be called "function".

4.4 Basics of Function and Complex Correspondence Detection

4.4.1 Introduction

Detecting complex correspondences between schemas, as well as functions within a correspondence, is a rather complex and difficult process. Since there has been hardly any research yet, we developed most strategies of our own, which was, of course, more difficult than implementing some already existing strategies. In this section, the basics of function and complex correspondence detection will be ex-

plained, which is necessary to understand the following sections where the exact implementation of detection algorithms will be presented.

4.4.2 Detecting Complex Correspondences

Detecting complex correspondences is very expensive if one uses the same strategy as detecting simple correspondences. In this case, all $2^n \cdot 2^m$ combinations have to be checked, where n is the number of source nodes and m the number of target nodes [25]. This means an exponential complexity, entailing unacceptable execution times for larger scenarios (remember that detecting simple correspondences requires only $n \cdot m$ comparisons). Besides this, for each correspondence exist several connector functions which could be used [15]. It was therefore much more appropriate to concentrate on another strategy which would be less complex. However, this also implies that not the entire search space is covered anymore.

To detect complex correspondences, we use the coma match result matrix as a base, so the set of (1:1)-correspondences detected by the COMA++ matchers. We analyze this matrix and try to discover complex correspondences, which we add to the alignment. This also means that the program will be executed when the COMA++ matchers have already finished, so when a first map (a set of simple correspondences) is already available. The program regards these simple correspondences closer, analyzes source and target schema, checks instance data (if available), and this way produces complex correspondences out of the original (1:1)-correspondences. Detecting complex correspondences is therefore a kind of mapping refinement, which takes part after the actual matching process is completed. This way, the complex correspondence detection became much easier, because we already have a list of simple correspondences. On the other hand, the dependency to COMA++ (resp. the correspondences it detects) is very strong.

4.4.3 Detecting Functions

Detecting functions is less expensive, because in the first version we only allow at most one function per correspondence. Furthermore, functions must be part of a correspondence and cannot appear alone. With regard to the complexity, we have to check for each correspondence a list of functions that might be needed there. This means quadratic complexity and should be quite acceptable.

Compared to complex correspondence detection, function detection works much different and is a complete separate thing. For this, the match result matrix is not important anymore, but instance data will be widely exploited. In fact, all functions which can be detected are based on instance data, which is analyzed and evaluated. This way, some functions that seem necessary can be added automatically to a correspondence. Another way would be to analyze meta-data of the schema elements that participate in a correspondence, such as constraints, data types, etc. Of course, these data types are not always available, and they are not always reliable

parameters to decide whether a function might be necessary or not. For the first version we therefore disregarded meta-data, also because of the large amount of schema types COMA++ supports (which otherwise would have resulted in a large amount of different constraint representations).

4.4.4 Distinguishing Functions and Complex Correspondences

The terms *function* and *complex correspondence* express completely different things, but in this study they are often used in the same context. The main point is that a complex correspondence always needs a function so that complex correspondence detection automatically includes function detection. However, in this study we try to differ between those two aspects, because from the implementation point of view they are quite different. As already mentioned, complex correspondence detection means analyzing existing (1:1)-correspondences and the schema nodes (whereas instance data is rather a minor matter). On the contrary, detecting functions means analyzing instance data (whereas the correspondences and schemas themselves are rather a minor matter).

Eventually, we say that complex correspondence detection means detecting complex correspondences automatically, including the correct connector functions. Function detection means detecting general functions between (1:1)-correspondences, like *lowercase*, *trim*, *round*, etc.

4.4.5 General Workflow

The complex correspondence detection and function detection works as follows: the program gets the match result matrix (the list of simple correspondences) when the COMA++ matchers have finished. After this, these (1:1)-correspondences will be included in the alignment list of the new data structure – remember that this alignment expresses the same as the match result matrix does, just in a more complex structure.

Subsequently, all correspondences will be iterated consecutively, and for each correspondence several strategies will be exploited. It will first be analyzed whether the (1:1)-correspondence could be part of an actual complex correspondence, and if so, the complex correspondence will be generated. Second, it will be checked whether there is a function which might be needed for this correspondence. Since we allow only one function per correspondence at the moment, a function can only be added to a (1:1)-correspondence. After this is accomplished, the next correspondence is analyzed and so forth.

For the complex correspondences, and for each function which can be detected, exists one more or less complex strategy which gets a (1:1)-correspondence and checks whether the correspondence seems to be a complex one or whether adding a function seems necessary. The strategies return the untouched correspondence if this

is not the case, or if it is, they return the enhanced correspondence. The enhanced correspondence will be added to the alignment then, and the original one (the simple correspondence detected by COMA++) will be removed to avoid inconsistency and redundancy. However, the match result matrix is never altered so that all original (1:1)-correspondences can still be accessed if this should be desired; even then, when the complex correspondence and function detection process is over.

If a correspondence has already a function, and another strategy detects a further function which might be required, we currently meet with a little problem, because for the first implementation only one function per correspondence is allowed. In this case, the function is only added if its priority is higher than the priority of the other function. However, when the program is extended to deal with more than one function per correspondence, the second function would be added, of course.

After each correspondence was checked one time by the different strategies, some other algorithms will be run to detect further complex correspondences. While the strategy introduced above considers only one correspondence at a time, and tries to prove that this simple correspondence is actually a complex one, the second strategy considers several correspondences at a time. It counts the number of correspondences between a source and a target element, and with due regard to the node levels where these elements occur, tries to prove that those correspondences actually form a complex match. This second strategy is called *Extended Complex Correspondence Detection* and refers to scenario 2 and 3, which will be explained in section 4.5.

After this is accomplished, the *Pattern Algorithm* can be run (described in section 4.7), and at last the complex correspondence and function detection is finished. The result will be returned to COMA++.

4.4.6 Dependencies

There exists a strong dependency to COMA++, especially for the complex correspondence detection. Since complex correspondences are detected out of simple correspondences, a match result matrix containing many false correspondences could impair the detection strategies considerably and cause false complex correspondences. The function detectors depend less on the match result, because they mainly concentrate on instance data, but operate on the simple correspondences nonetheless.

On the other hand, COMA++ is completely independent from the complex correspondence and function detection module. If, for any reason, the complex correspondence and function detection should not be executed, it is very easy to obtain this. In this case, only the call-operator for the detection module has to be disabled, and COMA++ would work as ever.

4.4.7 Further steps

When the map is generated by the program, the user can manually change it until it fulfills the expected requirements. With the mapping part accomplished then, other tasks can be carried out, e.g., data transformation or transformation script generation; however, these steps are not part of this thesis.

4.5 Complex Correspondence Detection

4.5.1 Introduction

The main idea of detecting complex correspondences out of (1:1)-correspondences is the following: each schema can be represented as a tree, and we would only expect data objects in the leaf nodes. If there is a node *name* that has two subnodes $firstname$ and *lastname*, we generally assume to find name data in these subnodes, and would ignore the *name* node with regard to data transformation. Correspondences where at least one element is an inner node would be therefore pointless relations (as it was already described in section 2.2.4), but of course, COMA++ also detects relationships of this type. As already explained, such correspondences are not useless per se, even though they are useless for data transformation; they may help considerably to find the correct correspondences.

Our first and major strategy was therefore the following: a (1:1)-correspondence between an inner node and a leaf is not possible, but there must be some relation between those two elements anyway (because there must be a reason why COMA++ detected this correspondence). If we now grasp the subnodes of the inner node (the leaf nodes), connect them and map them to the leaf node of the opposite schema, we suddenly have a complex correspondence, and only the leaf nodes are connected (the nodes where we expect data). This correspondence is not created randomly, but is justified to a certain degree, because there has been a relationship between the inner node and the leaf node after all.

We implemented this strategy first, and found out that this way complex correspondences like $((firstname,\ lastname),\ name)$ were easily detected out of the original (1:1)-correspondence $(name(firstname, lastname), name)$, so we already could deal with the standard example of complex correspondences. However, we also found out that there are many sub-cases and similar cases which we had not regarded so far. These cases will be described in the next section.

4.5.2 Dealing with different cases

The picture below shows 10 scenarios of complex correspondences which are represented by (1:1)-correspondences. Each scenario consists of 4 figures. Figure a) and b) show how the "complex correspondences" are expressed by (1:1)-correspondences, so the way how COMA++ would perhaps express them. Figure a) describes the case (1:n) and figure b) describes the case (n:1). Figure aa) and bb) show what the correct complex correspondence would be, so what kind of correspondence our program must derive from the given schema a) resp. b). To explain these scenarios, the case (1:n), so case a) and aa), will be used, but for the case (n:1) it is analogous.

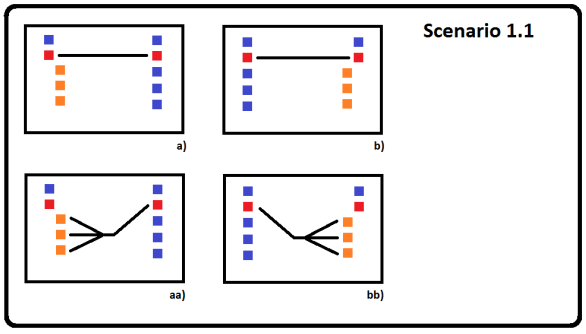
Scenario 1.1 is the simplest scenario, and was also the first case we implemented. It shows a (1:1)-correspondence between an inner node and a leaf node. Since no data is stored in inner nodes, but certainly in the leaf nodes (so in the subnodes of the inner node), the complex correspondence will be created out of the subnodes of this inner node. The overhead (1:1)-correspondence will be removed, of course.

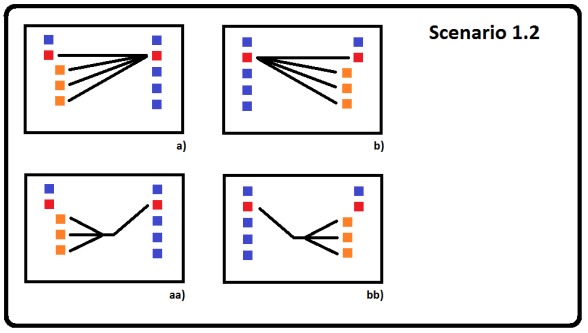
Scenario 1.2 is almost the same as scenario 1.1, except that the leaf nodes are already connected with the opposite node. In this case, the procedure is very similar, but now all (1:1)-correspondences have to be removed first, and after this the complex correspondence has to be generated. This scenario might occur pretty often; 3 simple correspondences could be, for instance, ($name$, $name$), ($name$, $firstname$) and ($name$, $lastname$).

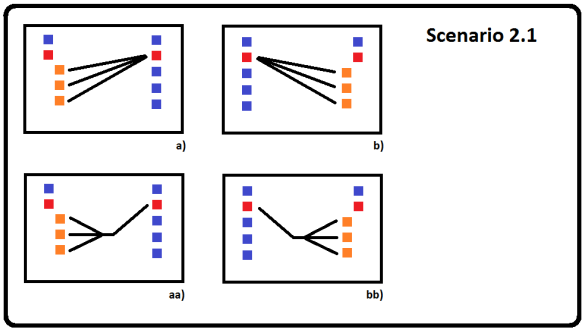
Scenario 2 is more difficult now, because the overhead correspondence is missing. Looking for (1:1)-correspondences, where one node is a leaf node and the other node is an inner node, is therefore not possible, and thus our main strategy does not work here anymore. However, the structure of the schemas can still be exploited to justify a complex correspondence, even though the likelihood for a true complex correspondence seems generally lower than in scenario 1.

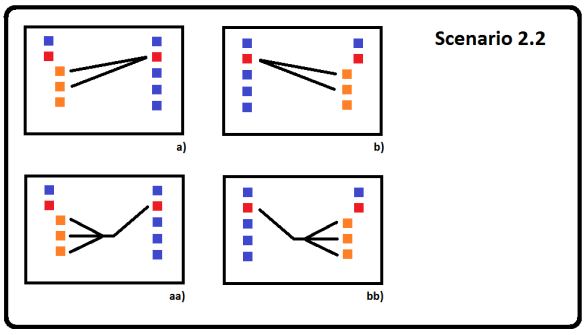
In **Scenario 2.1** all subnodes of an inner node correspond to a leaf, so in this case the program must check whether all subnodes of an inner source node correspond to the same target node. If this is the case, the complex correspondence will be generated out of these nodes. The likelihood for being a true complex correspondence seems, like in scenario 1.1 and 1.2, pretty high.

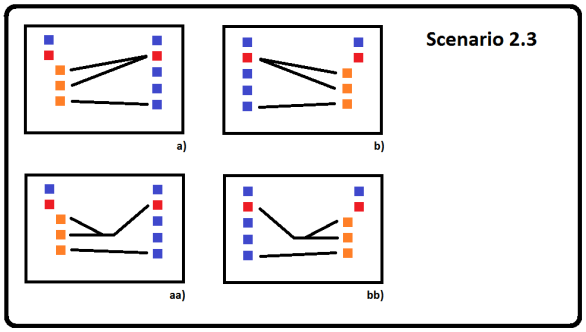
In **Scenario 2.2** only a part of the source subnodes corresponds to the leaf node. In this case we have basically two opportunities: only connect the nodes to a complex correspondence which are linked to the target element, or simply connect all source subnodes and map them to the target element. The first solution seems more cautious; we completely ignore the nodes that do not correspond to the target node, although they all have the same father node. The second solution seems bolder. We say that all subnodes simply correspond to the target node, because they have a unique father element – we do not regard the fact that some subnodes are not in a relationship with the other target node. In this second case, we assume that

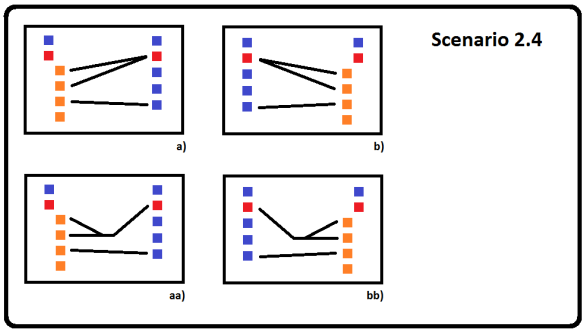

Scenario 1.1
a)
b)
aa)
bb)


Scenario 1.2
a)
b)
aa)
bb)


Scenario 2.1
a)
b)
aa)
bb)


Scenario 2.2
a)
b)
aa)
bb)


Scenario 2.3
a)
b)
aa)
bb)


Scenario 2.4
a)
b)
aa)
bb)

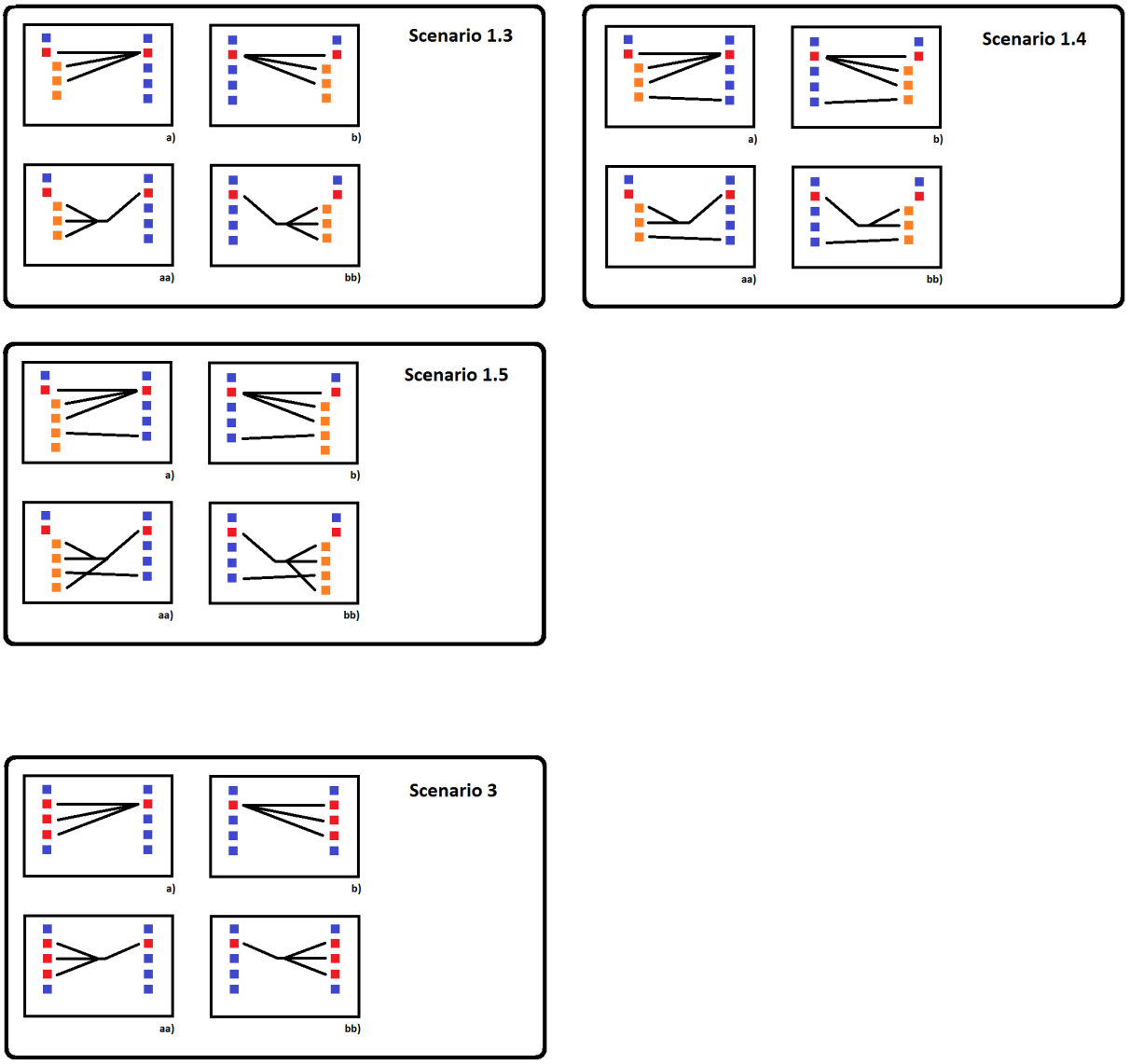


Figure 4.3: The 10 scenarios we considered for complex correspondence detection. The scenarios are ordered in the way they are described in the study.

COMA++ is not able to detect all true correspondences (some assumption which would apply to every schema matcher), and this way justify the inclusion of the not-connected subnodes into the complex correspondence. Facing these two contrary opportunities, we implemented a solution which is between the two cases. We say, if only one subnode is not connected with the target node, we include it into the correspondence (it might be an "outlier" which was not detected by the schema matchers). However, if more than one inner node is not connected with the target node, connecting all subnodes to the target node seems too risky, and the complex correspondence will not include the not-connected nodes. Of course, this thresh-

old could be increased or reduced, but we found 1 a rather sensible value for the beginning.

In **Scenario 2.3** all subnodes are connected with target elements, but take part in different correspondences. In this case, the nodes which are not connected with the main target element will keep untouched – the other nodes will be included into a complex correspondence. This scenario is practically a special case of scenario 2.2. We do not bother about the source nodes which are not related to the main target element, because they refer to other target elements, and thus already have a partner.

In **Scenario 2.4** some nodes correspond to the main target node, others take part in other correspondences (that is, refer to other target nodes), and others again are in no relationship (so, they are "free"). In this case, we connect the source elements which match the main target node and build a complex correspondence. The subnodes which are in another relationship keep untouched again, as well as the free subnodes which have no partner. Now the question might arise, why we do not use a threshold of 1 again, thus including a free subnode in the complex correspondence if it is the only free subnode. Our idea was, that unlike in scenario 2.2, we are facing two problems here: first of all, it is not sure whether this node really has to take part in one of the correspondences (that's the same problem as in scenario 2.2.), and second of all, we have to deal with more than one correspondence here (that is a problem that does not occur in scenario 2.2). Therefore, we would be forced to make a decision to which correspondence we think this node belongs, and that strategy seems too fuzzy to expect successful mappings. Hence, all free nodes keep untouched here.

After regarding scenario 2.2, 2.3 and 2.4, it became clear that these cases also apply to scenario 1, so in the case of an existing overhead correspondence. Therefore, 3 further scenarios had to be regarded (1.3, 1.4 and 1.5). However, since the overhead correspondence in scenario 1 is considered rather strong (meaning a high confidence for a complex correspondence), we simply do the following: we connect all subnodes and include them into the complex correspondence, whether they refer to the main target node or not. Only if one of the subnodes already corresponds to another node (so, not to the main target node, but to another node in the target schema), we do the same as in scenario 2.3 and 2.4: this node keeps untouched, because this (1:1)-correspondence is considered more certain. As for the rest of the subnodes, they will be included in the complex correspondence.

Scenario 3 is quite similar compared to scenario 1.2, but we have no superior correspondence and no superior node which would justify the correspondence (so we have no structural justification any longer). We simply have to deal with a set of source nodes that match one target node. Nonetheless, the complex correspondence would be created here, but the confidence is even lower than in scenario 2. It could be the case that it is just a (1:1)-correspondence after all, where one correspondence is correct and the other is false (we always have to remember that COMA++ might detect false correspondences).

Having these 10 scenarios seems already pretty much, especially since some cases are special cases (subcases) of other ones, but making these distinctions offered us to perform a more precise complex correspondence detection. Besides, there are some further scenarios and sub-scenarios that we did not explicitly distinguish for the first version, mainly because the 10 scenarios made the program already pretty extensive. As for the implementation, most strategies cover several scenarios at a time so that the program is less complicate than it might appear at first glance.

It must be remembered here that for all cases two directions have to be regarded, so both the case (n:1), which requires the concatenation of subnodes, and the case (1:n), which requires splitting a node value and distributing its substrings among a set of nodes. We expected that these two implementations work analogous to each other for each scenario. That means that if the program detects a (1:n)-correspondence according to one of the scenarios, it would also detect the analogous (n:1)-correspondence if we mirror the map, that is, if we swap source and target schema and source nodes and target nodes of the correspondences.

4.5.3 Implementation Details

Scenario 1 was rather easy to implement. The program iterates all correspondences, and for each correspondence checks for the source and the target node, whether it is a leaf node or not. If exactly one of the nodes is an inner node, the program considers this a prospective complex correspondence. Then the subnodes of the inner nodes are taken and concatenated or split. Of course, for each subnode it has to be checked whether it is already part of another correspondence, and if so, it has to be excluded from the complex correspondence. Also the threshold of the maximum number of subnodes has to be regarded (as described later). As for sub-subnodes, they are included in the complex correspondence as well.

Scenario 2 was harder to implement, because the overhead correspondence, which was like a guide, is now not available. Here we had to check for each source node which was connected with a target node, whether it has a superior father node. If so, we grasped all subnodes of the father node, and checked how many of them correspond to this unique target node. Then we had to decide whether we create the complex correspondence or not, according to the scenarios and the thresholds. After this, we had to go the other way around, and check for each target node whether there is a father node which might suggest a complex correspondence. We had to be very careful here so that a correspondence is not created twice or more.

In scenario 3 we have to iterate all source nodes, and for each source node have to grasp all target nodes which are connected with it. If a source node is connected to more than one target node, and none of the target nodes is connected with any other source node, the complex correspondence will be created. As matter of fact, the other way around has to be regarded as well.

Especially scenario 2 and 3 require high time complexity, because many node com-

parisons and correspondence iterations have to be performed. Besides, we have to be very careful to avoid duplex correspondence detection.

4.5.4 General Obstacles

Introduction

So far, detecting complex correspondences might appear not as difficult as one might expect first, but truth is that there are several problems connected with the entire process, which reveal themselves only after further consideration.

At first, detecting a complex correspondence means to detect which source and target elements of the map correspond, but after this is done, further aspects have to be regarded. We also have to detect...

- the correct connector function,
- the order of the elements, e.g., whether it is $concat(firstname,\ "\ ", lastname,\ "\ ",\ title)$ or $concat(title,\ "\ ", firstname,\ "\ ",\ lastname)$,
- the separator (only if we deal with string-based correspondences).

Besides this, some other problems occur and will be explained in this section as well.

Determining the Correct Connector Function

The first problem that occurs if a complex correspondence is finally created, is to determine the correct connector function. For strings, the functions *concatenation* and *split* are normally used; concatenation for (n:1)-correspondences and split for (1:n)-correspondences. In most cases, a separator is needed as well, e.g., if names are concatenated, a blank must be put between each name string. If the split function is used, a separator has to be used again, telling the program where to split the string. Simply creating a concatenation or split function without using separators, as described in some papers (e.g., [29]), is the easiest way, but not sufficient in practice.

Complex correspondences between numerical nodes require at least one numerical function, but normally even a set of numerical functions, like $price = basePrice * (1 + brokerage)$. Detecting such functions seems extremely difficult. In iMAP the data objects are regarded, the scope and distribution is ascertained and then different combinations of arithmetic functions are created and tested against the scope and distribution. The combination that comes closest to the scope and distribution is considered the right one then. We tested this strategy as well, but only got few correct functions, mainly because iMAP uses much more complex mathematical strategies. We therefore decided to only generate an empty "formula function"

where the user has to specify the correct arithmetic expression. Thus, the program detects complex correspondences between numerical elements, but does not detect the correct expression (some drawback we had to accept for the first version).

If there is a mixture of numerical nodes and string nodes in one complex correspondence, one has to decide whether one uses a string connector (concat/split) or a numerical one. However, a string connector seems much more appropriate then, because numbers can widely occur in strings (like in address elements), but non-numerical characters like letters typically does not occur in number elements.

Distinguishing between those two types of complex correspondences (string and numerical ones) is very important, but can sometimes produce the wrong kind of connector function. A classic example would be the correspondence ($telephone$, $telephone(areaCode,\ telephoneNumber)$). Since all elements contain numbers, the program would consider this a numerical complex correspondence, but it actually is a string one, so we would need a connector function like $concat(areaCode,\ telephoneNumber)$. Exploiting schema meta data (like data types) and checking number elements closer (e.g., if some numbers start with a zero) might be something for future work, and could still improve the connector function detection.

The Number of Source and Target Nodes

Sometimes it may occur that there is a link between an inner source node and a leaf node (scenario 1.1), and the inner node has a large amount of subnodes, which are all leaves. Let us assume there are 12 subnodes and all are strings. Now we would have a (12:1)-correspondence and actually concatenate 12 subnodes, mapping the result to the single target node. It is certainly very unlikely that concatenating 12 values would give us the correct target value, and it rather seems that this cannot be the correct complex correspondence. In this case, it would be wiser to avoid creating the complex correspondence.

For this case exists a special threshold. If the number of source nodes or target nodes of the supposedly complex correspondence exceeds this threshold, the complex correspondence will not be created. Of course, one obstacle is to determine which threshold is most appropriate. We used 5 by default, but maybe a larger value (6 or 7) would be quite appropriate, too. A value below 5 seems very risky, because a correspondence like ($address$, $address(street,\ no,\ city,\ zip,\ country)$) would already be rejected then.

The Separator

Another important step is to detect the correct separator between strings which are to be concatenated or which are to be split. For example, the function $concatenate(title,"\ ",firstname,"\ ",lastname)$ would use two blanks as separators between the element references. The function $split(date,\ "-",day,\ "-",month,\ year)$ would

have two dashes as separator for the date element. These separators are important for data transformation, and should be detected in addition to the complex correspondence and connector function.

If no instance data is available, there is practically no possibility to detect the separator, and thus a default separator is the only possible solution. It seems sensible to use a blank character as default separator, because it might have the largest probability to be the correct one.

However, if instance data exists, the program tries to detect the separators itself, whether it is the blank character or not. For this, we have a list of five possible separators (blank, dot, dash, slash and underscore), which we use on the instance data and thus try to find the correct separator.

If we have an (n:1)-correspondence, n source elements match one target element, so the concatenation function has to be created. By default, we would put a blank between the several n source values, however, if we have instance data, we first grasp all values from the target node and split them consecutively by the several possible separators. For each iteration we count how many substrings result by splitting the target values this way. If a target value will be split in exactly n subvalues (so the number of source elements of the complex correspondence), this separator can be seen as a real match; if not, it is a mismatch. The separator which gets the most matches seems to be the correct one, however, we say that a threshold of at least 50 % must be reached to denote the separator as the true one. If this value is not reached, the default separator will be used, although it is possibly the wrong one.

For example, we have the correspondence $((title,\ firstname,\ lastname),\ name)$, and therefore use the connector function $concat(title,\ ?, firstname,\ ?, lastname)$ (the question marks represent the separators that are to be determined). Then we grasp all name objects first and consecutively split them by blank, dot, dash, slash and underscore. We count how many names can be split exactly in 3 substrings, and might found out that 71 % of the names do so if we use the blank as split argument, whereas the other split arguments would get us far less, in most cases even 0 % evidence. Thus, we would consider the blank as the correct separator between the element references.

In the case of a (1:n)-correspondence, the split function has to be assigned. The strategy to detect the separator works analogously. Now all source values will be grasped, and then all separators are used to split the several source values in substrings. The separator which splits most source values in n subvalues (so the number of target nodes) is considered to be the correct separator again. Of course, the threshold of 50 % applies to this case as well.

The algorithm is suboptimal, because it only considers one separator for several subnodes. This should work for a lot of examples (like title + first name + last name), where the blank separator is used between all substrings, but in some cases different separators would be necessary for one correspondence. It can also occur

that a separator consists of more than one character, like $concat(street, ", ", zip, " ", city)$. For this, the strategy does not work either.

Still, for our first solution the simple strategy seems rather sufficient, also because of the larger complexity checking all combinations of separators would entail. Besides, the user is always able to change a separator manually, so that all possible cases of concatenations and splits can be performed in the program.

More Separator Problems

Sometimes the split function does not work for all data objects, and sometimes it works only for the minority, even if the correct separators have been detected or determined by the user. This happens if the data objects consist of a different number of separable substrings, like name objects often do. If an element combination $(firstname, lastname)$ exists, and we create the correspondence $(name, split(name, firstname, " ", lastname))$, the objects "Peter White" and "Lucy Collar" would work, but objects like "George Wilhelm Bush" or "Cathy Bartash of Canterburry" would not. Dealing with street names in address strings like "Oxford Road", "N Miami Ave." and "W St. Johns Dr." pose an even more difficult problem. Without background knowledge the transformation would go wrong here.

At this point, it has to be remarked that the concatenation function always works fine, and is far less error-prone than the split function, where each string has to be split into the very exact number of subnodes. Referring to the example above, it does not matter how street names are structured, and how many blank characters they consist of – concatenating these strings will cause no trouble. However, splitting strings incorrectly would lead to the fact that the data object cannot be transformed (or would be transformed incorrectly, something that is even worse). Since both cases normally occur at the same rate, we have to expect that 50 % of all complex string correspondences are of the type (1:n), and would thus require the aggravating split function.

The Order of Subelements

One further problem which always occurs when dealing with complex string correspondences, is to detect the correct order of subnodes resp. substrings ("subvalues") when concatenating nodes resp. splitting node values. By default, we would expect that the order given in the schema is the correct one, because schemas are normally designed manually, and one would assume that the order of nodes is rather intuitive. In many cases this is indeed true, and we have correspondences like $(name(title, firstname, lastname), name)$ where the order of subnodes is already correct. However, it may also happen that the order is not correct and we have to deal with correspondences like $(name(firstname, lastname, title), name)$. In this case, we want to detect that the title is normally at the beginning of the *name* ele-

ment, and the correct correspondence would therefore be ($concat(title,\ firstName,\ lastName),\ name$).

We implemented some rather complex strategy to detect the right order of subnodes (in case of the concatenation function) resp. subvalues (in case of the split function), but again depend much on instance data. The idea is the following: let us assume we have an (n:1)-correspondence (($firstname,\ lastname,\ title),\ name$), and we already know the correct separator. We then grasp all source values (so all firstname, lastname and title objects) as well as all target values (so all name objects). We split each target value by the separator (which we must know, of course), getting the respective substrings and compare them with the subnodes. We have now kind of an "(n:n)-matrix" (we discard all target values which cannot be divided in the exact number of source elements). We call the substrings of the target values *subvalues* and will name them in this example as $name\_sub1$, $name\_sub2$, $name\_sub3$; the source nodes (subnodes) we still call $firstname$, $lastname$ and $title$. We can now create an (n*n)-matrix, and compare each subvalue with each subnode. Analyzing the values closer, could help us to fathom out which subvalue belongs to which subnode.

In point of fact, we even create 3 (n*n)-matrices, each regarding different aspects. Matrix 1 specifies how many source node values also appear in the target subvalues. So we start to iterate the elements of the node $firstname$. The first value we might find there could be "Paul". Now we check sequentially how often "Paul" appears in $name\_sub1$, $name\_sub2$ and $name\_sub3$ (remember that we do not know yet which of the 3 subvalues is the first name value). It might appear 0 times in $name\_sub1$, 11 times in $name\_sub2$ and 0 times in $name\_sub3$. We write (0, 11, 0) in the first row of the matrix and go on checking the next name of the subnode $firstname$. This might be "John" now. Again, we look how often this value appears in the 3 subvalues of the target element and might get the result (1, 17, 0). The first row will be updated to (1, 28, 0). This already shows that there is a large similarity between the first subnode and the second target subvalue (so between the source element $firstname$ and $name\_sub2$, so the second subvalue of the target element $name$). We go on until all first names are iterated, or a special threshold is reached to quicken the entire process. After this, we go on with the next subnode element, which is $lastname$. Now we iterate all last names and check how often each value occurs in $name\_sub1$, $name\_sub2$ and $name\_sub3$ again, etc. This way, the second row of the matrix is initialized.

At the end, we have a fully initialized matrix (see also the picture below), which might suggest the correct order of the subnodes or subvalues. We get the right order of a subnode resp. subvalue by searching for the maximum value in a row resp. column. In the first matrix presented below, we have the correspondences ($firstname, sub\_name2$), ($lastname, sub\_name3$) and ($title, sub\_name1$). This order (2, 3, 1) has to be put in the correct order now, so 1, 2, 3. That implies the function $concat(title,\ firstname,\ lastname)$.

However, there is one problem connected with this first technique: it might happen

	$Value_1$	$Value_2$	$Value_3$
$FirstName$	0	**1374**	29
$LastName$	0	29	**308**
$Title$	**571**	0	0

	$Value_1$	$Value_2$	$Value_3$
$FirstName$	0	**41**	5
$LastName$	0	5	**14**
$Title$	**26**	0	0

	$Value_1$	$Value_2$	$Value_3$
$FirstName$	2.7	**1.3**	1.7
$LastName$	3.1	**1.7**	1.9
$Title$	0.3	2.7	3.1

Table 4.1: The three tables used in the program to judge the similarity between sets of strings. The first matrix compares the number of duplicates between the sets (accumulative), the second matrix compares the only duplicates (not accumulative), and the third matrix compares the deviation of the average string length between the sets of string values.

that the correct order cannot be detected, e.g., because (1,1) and (2,1) are both maximums of a matrix, but of course only one subnode (subnode 1 or subnode 2) can be the first one. Thus, to detect the correct order, the maximums of the rows must be in distinctive columns (and the maximums of the columns in distinctive rows), which we call a *clear assignment*, meaning that the order of subnodes resp. subvalues is unique according to the matrix. We also found out that the first idea was good, but not always perfect. If the names John and Paul occur frequently, a large value would be the result, although they might match the subvalues only by accident (remember that the names Paul and John were actually only 2 matches, but got us already 28 credits). Therefore, we developed two further matrices to help us.

Matrix 2 only stores the unique matches, so instead of getting (0, 11, 0) and (1, 17, 0) like in the first case, we get (0, 1, 0) and (1, 1, 0) here. After the second value check, we end up in (1, 2, 0) instead of (1, 28, 0). In this case, unique value matches are considered more important than the number of total matches, however, this algorithm does not consider duplicate values. If the name "Peter" occurs three times in the source schema, it would get 3 credits instead of 1. A list of names which have already been accepted would help to avoid this little drawback, but would also increase the complexity.

Matrix 3 finally compares the average string lengths between the element combinations. A value of 0 means that both source value and target subvalue have the very exact average value length, and therefore seem very similar. A large value like 5

means that the values of the one node are considerably longer (resp. shorter) than the values of the other node ("subvalue") – they seem not very similar. The difference between $(firstname, firstname)$ would therefore be rather small, as well as the difference between $(lastname, lastname)$ (if we have a very large amount of data objects, they would even converge to 0). $(title, title)$ would also have a low value, whereas $(title, firstName)$ would certainly have a larger value, because titles are normally short strings, whereas names are considerably longer. Thus, the position of the title element could be pointed out without checking duplicates, but by comparing string lengths. However, this measure is not a panacea – $(firstname, lastname)$ would probably have a low value as well, and thus could tend to be considered equal.

These 3 matrices are initialized in one step (so in parallel) to save execution time. After this, the first matrix is checked. If it has a clear assignment (so the correct order can be exactly determined), we order the elements according to this matrix. If not, we try the second or the third matrix. If neither matrix has a clear assignment (this is already the case if all matrices contain zeros), we return null, telling the program that the order cannot be detected.

The sketch above shows how the 3 matrices of the $((firstName,\ lastName,\ title),\ name)$ correspondence could look after they were initialized. The maximums (matches) are printed bold. Thus, it can be seen at once that matrix 3 has no clear assignment. However, this would pose no problem, because the first matrix has a clear assignment (and the second matrix as well), and would be used to create the right order of elements. The problem in matrix 3 occurs, because the average length of last names in the second row is slightly closer to the average length of first names in the opposite schema, although the combination $(lastname, lastname)$ would be correct here. Due to this false assignment, we have no clear assignment any longer, and could not use this matrix for determining the correct order of elements.

Adjusting the Split function

The functions *concatenation* and *split* are the most important functions, because they are needed for complex string correspondences. Since we also regard the order of subelements, and invented a whole strategy to detect it, we soon found out that the classic split function did not work here (whereas the classic concatenation function works fine).

The classic split function, as known from programming languages, deals with two parameters: a string and a separator. It returns a list of substrings created by using this separator. Thus, the function $split(\text{"}Dr.\ William\ Scott\text{"},\ \text{" "})$ would return $Dr.$, $William$ and $Scott$, just the way we expect and want it.

However, we soon encountered some problems: first, different separators could appear in a string, like in addresses where we have to deal with blanks and commas, and second, the order has to be assigned anyway. Therefore, we extended the split function as follows: $split := split(sourcenode,\ [separator,\ targetnode]+,\ targetnode)$.

Thus, the following example would be possible: $split(student.address, " ", st.street, " ", st.number, " ", st.zip, ", ", st.city, st.country)$.

This way, much more precise mappings can be created, although there is still a problem remaining which we cannot solve: $split(student.name, " ", st.firstname, st.lastname)$ would split strings like "Kevin James Scott" into "Kevin" and "James Scott" (and not in "Kevin James" and "Scott"). This is because the program searches for the first blank in the string, and will assign the front part to the first name and the remaining part to the last name.

Target Nodes Participating in Different Correspondences

As already described, a target node should never take part in different correspondences, because if it comes to data transformation, it can only take the value of exactly one correspondence. A map where a target node takes part in more than one schema must therefore be a false and invalid one (under the assumption that no filters are used; otherwise a target node can indeed appear in more than one correspondence).

In the complex correspondence detection we tried to prevent this case, but it may generally occur nonetheless – we deliberately decided not to prevent this case under all circumstances. The reasons for this was mainly the bad recall which would result in this rather strict assumption. Besides, an invalid scenario can be of great use for the user, because adjusting the correspondences is much easier than discovering them.

4.6 Detecting Functions

4.6.1 Introduction

A couple of functions can be detected automatically, while most of the possible functions cannot. Our primary goal was only to offer a distinguished selection of functions in COMA++, whereas the function detection was rather an additional part. Nonetheless we could implement some function detection strategies for several functions, which can support the mapping process considerably.

For the first version we implemented 15 functions which are enumerated in the table below. As matter of fact, adding further functions to the program is generally possible.

No	Name	Type	Description	Detection?
1	Concatenate	String	Concatenates two strings.	YES
2	Split	String	Splits a string into substrings according to split characters.	YES
3	Lowercase	String	Turns all letters of a string into lower case letters.	YES
4	Uppercase	String	Turns all letters of a string into upper case letters (capitals).	YES
5	Trim	String	Removes preceding and following blanks of a string.	YES
6	Substring	String	Extracts a substring from a given string by applying a start and end position.	NO
7	Replacement	String	Replaces a value by another value (multiple replacements are possible).	PARTLY
8	Add	Numerical	Adds two or more values.	NO
9	Subtract	Numerical	Subtracts one or more values from a value.	NO
10	Multiply	Numerical	Multiplies two or more values.	NO
11	Division	Numerical	Divides a value by one or more values.	NO
12	Round	Numerical	Rounds a value to an integer.	YES
13	DateConversion	Date	Converts a date format into another date format.	YES
14	CurrentDate	Date	Returns the current date.	NO
15	CurrentTime	Date	Returns the current time.	NO

Table 4.2: The 15 functions available in the new COMA++ version.

As it can been seen, the program offers far less functions than some commercial solutions like Microsoft BizTalk Server or Altova MapForce; however, most use cases can be covered with these few functions, because most of the functions in the proprietary solutions are special functions for very rare cases.

4.6.2 The Basics of Function Detection

Whereas implementing the functions itself was rather simple, the function detection became the main part of this Master's thesis besides the complex correspondence detection. There seems to be no program available today which would offer automatic function detection; maybe because it is rather complicated and error-prone after all. As it can be seen in the table, we still implemented strategies which can detect some functions automatically. They are sometimes based on heuristics (just as in the complex correspondence detection), but have normally a far better confidence than the automatically generated complex correspondences.

Function detection is performed by analyzing instance data connected with the source and target elements of the several correspondences, intending to find some hints for the necessity of a function. These algorithms detect only functions between (1:1)-correspondences, so we will not regard complex correspondences any longer.

For now, exploiting instance data seems to be the only way to detect functions. In some cases, using meta-data could also help to detect them, but as already mentioned before, we do not use meta-data at the moment. Thus, the function detection is possible, if, and only if, instance data for both schemas is provided. The function detection is therefore quite different from complex correspondence detection, because we will not regard elements and the schema structure anymore, but will entirely concentrate on instance data sets.

4.6.3 Detecting Functions in (1:1)-Correspondences

Introduction

The following functions can be detected automatically:

- lowercase and uppercase
- round
- trim
- date format

The basic principle is always the same: first analyze all values of the target element, and then analyze all values of the source element. If all values of the target element are lowercase (resp. uppercase, rounded, trimmed), but at least one value of the source element is not lowercase (resp. uppercase, rounded, trimmed), then create the lowercase (resp. uppercase, round, trim) function.

An example might be the following: there is a correspondence between the two elements *booktitle* and *title*. At first glance, the correspondence seems fine, but maybe one of the functions mentioned above might be necessary. This applies to the functions trim, lowercase and uppercase (the round-function is of no use if string elements are compared, and the date-format function shall be disregarded for now).

Now all instance data of each element is examined. Let us assume that we have the instance data of the source element *booktitle* and the target element *title* as depicted in the table above. Regarding the titles, the program recognizes that all values of the target schema element *title* are lowercase, and it therefore assumes that this is a general constraint of the element. Now it analyzes the values of the source schema element *booktitle*, and recognizes at once that they are not solely lowercase. The lowercase function seems quite necessary here, and will be therefore added immediately.

booktitle	*title*
The Secret Garden	gone with the wind
Anne of Green Gables	tale of two cities
The Railway Children	robinson crusoe
Huckleberry Fin	pride and prejudice
	jane eyre

Table 4.3: Instance data values of the source element *booktitle* and the target element *title*. After analyzing these values, the program would suggest the lowercase function.

For the other way around, so a mapping from *title* to *booktitle*, nothing is to be done. If all values of the source element would be lowercase, but some of the target element would not, this would be quite fine, because being lowercase is a weaker constraint than not being lowercase, and thus no function would be needed here (we always assume that the target database sets the constraints, not the source database). However, if regarding this case more closely, there could be a function even for this case, which would be a function converting the lowercase book titles so that each word starts with a capital, except stop words (and again except stop words that occur at the beginning of the string). Such a function is far too specific to be part of a function set a schema mapper could offer, but it might be a user-defined function.

At this point, it has to be remembered that a function between two elements only works for one direction (transformation from source schema to target schema). Regarding the other direction, they are not necessarily correct, nor would their supposedly inverse function (like uppercase) be correct; $uppercase(lowercase(title))$ is not necessarily the same as *title*. Since we never regard the direction target schema to source schema, this does not pose a problem, though.

The Round Function

Detecting the round function can be a little error-prone. If the program recognizes that all values of the target schema element are integers, but at least one of the source schema elements is not, the round function will be added. With this, the algorithm works just as before, but two problems might occur: first, there are other functions like ceiling and floor, which almost do the same as the round function does, yet the program cannot determine which of these three functions is the correct one. By default, the algorithm decides on the round function, because this function would probably occur most frequently, but it can never be sure whether this choice was the right one.

The second problem could occur if source and target elements have different measure units or different scopes. The two corresponding elements $(bodysize, bodysize)$ could use the measure units meter and centimeter for instance, so the one element

containing values like 1.85, 1.60, 1.72, 2.01, and the other element containing values like 174, 163, 190, 177. Offering the round function for the first element would be wrong in this case, because values expressed in meters must not be rounded here, but multiplied by 100. In this case, a wrong function would be detected (which at the end would convert all source values to 1 or 2). Analyzing the values closer, e.g., computing scope and distribution, could help to prevent this.

The Date-Conversion-Function

Dealing with different date formats is a widely occurring problem when databases are to be integrated. This problem especially emerges when databases from different countries are to be integrated, but there can be different date formats even within a country or region itself. Basically there are two classic types of dates: a short form and a long form. The long form like "Mar 18, 2011" or "18th March 2011" is rather rare in databases, but could appear nevertheless. However, for the first version we did not consider the long form, chiefly because this would have implicated to offer a list of month names and abbreviations (ideally in different languages like German, English, France, etc.), which would have led to a background knowledge approach. We also did not regard the time in a date string (like 2011-05-01, 15:36:03 UTC).

There is also a third date format, which is especially used in database systems. This format represents dates in milliseconds, specifying how many milliseconds have passed since January 1, 1970 [20]. This format will be neglected in the first version as well, but might be something to add in a later version.

For the short form of dates there exist different perspectives:

- The order of year, month and day.
- The separator between the three components (normally a dot, a dash or a slash).
- The usage of left-hand zeros (e.g., 3/18/2011 or 03/18/2011).
- The year format, which can be long or short (e.g., 2011 or 11).

Besides these issues, it is not always the case that day, month and year are represented. Sometimes only day and month might be stored (e.g., for a birthday or holiday calendar), whereas in other cases only month and year are stored (e.g., for the edition date of magazines).

Detecting functions to convert date formats is also an instance-based algorithm, but is much more complex than discovering the trim, round or lowercase function. For this, the instance data of the source and target element of each (1:1)-correspondence is examined. We therefore developed a method which knows all common date formats of the short form. If a string like "18.03.2011" or "3/18/11" or even "2011-03-18" appears, the function is able to recognize that this is a date, and even finds

the correct date format (like "DD.MM.YYYY", "M/DD/YYYY" and "YYYY-MM-DD"). If all values are a date and have a unique pattern (which is usually to be assumed in a database), the method returns the respective pattern. This is done both for the source and the target element of the correspondence. If these two patterns are not equal, the date-conversion function will be created, which gets the pattern of the source schema and of the target schema as input – from this point of view, the date conversion function is detected in the same way like the other functions. Finally, when transforming data, the transformation algorithm can convert the date format according to this pattern.

This algorithm works fine in common databases, especially if there is enough instance data. To distinguish day and month, the function searches for values larger than 12. Problems occur if no days occur that are larger than 12, if wrong dates exist (like 14-20-2011), or if a rare date format is used (especially the formats YY-MM-DD, YY-DD-MM, DD-YY-MM, MM-YY-DD). However, all three cases will occur extremely seldom and do not occur in typical databases. Only the first reason is a little more likely: if dates are always generated at the first of a month (or even year), the program has difficulties to decide which part of the string is the day and which is the month. In this case a wrong format could be generated.

4.6.4 Advanced Function Detection: the Replacement Function

Introduction

Whereas functions like lowercase, uppercase, trim and round can be detected rather easily, the replacement function is much more complex, and unfortunately also more error-prone. However, under certain circumstances it can perform a pretty great job, and was therefore accepted to the function detection module of COMA++ nonetheless.

The classic replacement or translation function gets two strings A and B as its input, meaning that each string that matches A must be replaced by B. It is one of the most important string functions next to the concatenation and split (resp. substring) function.

Replacing some values of a node is a rather common task. In the 2nd chapter some examples were already given: "U.S." has to be replaced by "USA", "Mr." by "Mr", "€" by "EUR", "Information Technology" by "Computer Science", "Automobile" by "Vehicle", "Colour" by "Color" and so forth. The goal is always to use a unique vocabulary in the target database, which can be achieved by adding the replacement function to a correspondence. The reasons for those differences between source and target instances are generally of linguistic nature: synonyms, homonyms, different forms of abbreviations, imprecise usage of vocabulary and different spellings of words seem to be the most frequent ones. As for the semantic aspect, it is normally not to be changed, so replacements like $(book, movie)$, $(small, expensive)$ or $(male, female)$ would be quite unreasonable.

As described above, the classic replacement function normally gets exactly two parameters as its input. However, functions in schema mapping are sometimes a little different from mathematical functions or functions in programming languages, and may have slightly different aims, too. This was the case when we implemented the replacement function – we changed it in a way that it has an unbounded number of parameters, yet at least one parameter. Each parameter can be seen as a tuple (a, b), where a has to be replaced by b. This way it is possible to specify more than one replacement per correspondence.

An example would be the function *replacement*("$D :: Germany$", "$UK :: United\text{-}Kingdom$", "$FR :: France$", "$ES :: Spain$"). This would be necessary in a correspondence $(country, country)$ where the countries in the source database are represented by their international abbreviation, but in the target database by their name. As it can be seen, we use "::" as the separator between a and b, since it is rather unlikely that "::" itself would appear among the instance data.

Categories

Detecting the several replacements by using naïve methods only works for node sets which have few distinct values. These nodes are often categories. An important kind of categories is an element of the type boolean, so where only two different values are allowed. Such elements appear widely in database schemas, e.g., sex (male/female, m/f, 0/1, etc.) or marital status (single/married, s/m, 0/1, etc.). Also, general states like hasChildren, isFirstCustomer, isSolvent, etc. are typical elements which use exactly two different values (like yes/no, y/n, t/f, 0/1, etc.). In the following we call those categories *binary categories*.

Automatic Replacement Detection

There are functions which can be generally detected automatically, functions which can hardly be detected automatically, and functions which are right in between. The functions presented before (lowercase, date, etc.) belong to the first group, whereas most of the possible functions cannot be detected automatically (or only with an enormous effort). The replacement function occupies a special place in the function set, because it can be partly detected automatically, but altogether has a rather low confidence. We implemented it nonetheless, on the one hand because it was an interesting task, and on the other hand because it can be of much use for the user anyway.

The replacement function will be only taken into consideration at all, if, and only if, each source and target node of a (1:1)-correspondence has at most 5 distinct values (5 is a threshold which can be adjusted in the program). For example, if the only distinct values of a source node A are "red, cyan, yellow", and of the corresponding target node B "red, blue, yellow", the replacement function would be taken into consideration. On the contrary, having a value set like "red, green, purple, gray,

blue, yellow, pink, black, auburn, ..." in one of the nodes, it would not be possible to run the replacement function detector, because the number of distinct values is already above 5.

Now if this condition is fulfilled, the program tries to create an appropriate replacement function. For that, it fetches the distinct values from each node, counts how often each value appears, and orders the values by the number of their occurrences. This might look as follows.

Value	#Occurrences
Red	411
Cyan	288
Yellow	59

Value	#Occurrences
Red	78
Blue	69
Yellow	44

Table 4.4: The first table shows the distinct values and number of occurrences of the source element, the second table the values and occurrences of the corresponding target node.

In the following, the algorithm removes all values (rows) from the source and target table which are duplicates, i.e., which appear in both tables. Since they are equal, no replacement is necessary for them, and keeping them in the table would be useless. In the example above, this rule applies for the values *red* and *yellow*, which appear in both tables. However, instead of removing the rows entirely, we just set the values NULL so that the rank of the other nodes will not be impaired.

We therefore get:

Value	#Occurrences
NULL	NULL
Cyan	288
NULL	NULL

Value	#Occurrences
NULL	NULL
Blue	69
NULL	NULL

Table 4.5: The first table shows the distinct values and number of occurrences of the source element after the duplicates were removed, the second table shows the values and occurrences of the corresponding target node after the duplicates were removed.

The next step is to generate the replacement function. For this, each value of the source table which is not NULL is replaced by the value of the target table which has the same rank (and which is not NULL either). We therefore generate $replacement(\text{"}Cyan :: Blue\text{"})$.

Problems

As it can be seen, this algorithm depends much on an equal distribution of the values between source node and target node. As matter of fact, this cannot always be expected, and is a pretty bold assumption after all. The apparently correct assignment works in the above example, because both nodes have the same semantic ranking, so (red, cyan, yellow) resp. (red, blue, yellow) – their distribution is the same. If the ranking of the target node would be (cyan, red, yellow) instead, the replacement function would not be generated, because there would be "blue::NULL" then. Of course, such a replacement is pointless, and we would therefore do not generate it at all. In this case, the program could traverse the target values and search for values which have no partner yet. It would then create the replacement *blue* :: *cyan* (because they are the only values remaining). However, this solution seems too risky after all, and would especially produce errors if more than two values have no partner.

Even more risky is the next assumption the program makes: it always assumes that there exist the very same items in the source and target schema, just that they have different spellings. However, this is only scarcely the case, especially in non-binary categories. For instance, there could be two lists of a few cities which are not the same, like (Toronto, Vancouver, Ottawa) and (New York, Washington, Los Angeles, Chicago). Under these circumstances, the algorithm would produce replacements like ("Toronto::Chicago") etc., which would be completely wrong. Thus, the algorithm is especially error-prone if two distinct or partly distinct value sets exist (whose cardinality is below the threshold). Keeping the threshold low (at most 5, maybe even smaller) would therefore reduce the risk of creating wrong functions (higher precision), but would also reduce the recall.

On the contrary, the probability of creating wrong replacements between binary categories like in the correspondence (*gender*, *sex*) seems considerably lower. If two binary categories appear in one correspondence, it seems almost certain that both categories express the same things, and creating the replacement function, if possible, would be sensible. This would probably prevent the second problem (replacements between disjunctive sets, so pointless replacements), but the first problem (different distributions leading to wrong assignments) would still exist. If there work 57 % male employees in company A, and 38 % male employees in company B, the replacement function would create replacements like "m::female" and vice versa, only due to the different distribution. However, the program would at least detect that using the replacement function is definitely necessary in this case.

Since a user is able to discover a wrong function quickly, and removing it would only cost a few clicks, whereas discovering a missing function would be more complicate, offering the replacement function detection does not seem so bad after all, even if the generated replacements have generally a rather low confidence.

Prospect

The technique presented above is rather simple, and could be extended by further features to be less error-prone. Comparing the values on string-based methods could increase the confidence considerably; for this, classic methods known from schema matching could be exploited (e.g., string comparison methods etc.), which are already implemented in COMA++. Another idea for improvement would be to revise the ranking; instead of ranking it by the number of occurrences, exploiting more sophisticated distribution functions might perhaps reduce erroneous assignments as well.

4.6.5 General Obstacles

One general problem in function detection is that a constraint is erroneously discovered. For instance, there are 3 book titles which are all spelled lowercase, and the program creates the lowercase function because it assumes that lowercase is a constraint of this element. However, these 3 book titles could be lowercase by accident, and other books which are to be inserted have not necessarily to be lowercase. In this case, a wrong function would be added to the correspondence. However, the more instance values exist, the less likely it is that a constraint will be detected erroneously. Therefore, we have implemented a threshold again, which specifies how many data sets must exist so that the program would generate a function. This value is 5 by default, so if only 3 data sets would exist, nothing would be done. Note that 5 is the minimum threshold, so the number of values which must exist to run the function detector. If there are more values available, all will be checked, of course.

There is one further problem which might occasionally occur. In large databases it cannot be excluded that one or even a couple of instance values are wrong. For any reason a book like "the red Room" could be among the instance values which are actually all lowercase, or a garbled value like "384ANs9", or a wrong date (as already mentioned). In such case, the function would not be generated, or maybe a wrong function would be added to the correspondence. However, we basically expect to get correct data, and will always refer to this premise.

Thus, the main problem of function detection does not seem to be the detection process as such, but mainly the lack of instance data. If no source or target data exists, the entire function detection module cannot be run. Of course, it can be assumed that source data always exists, because otherwise nothing could be translated after schema mapping, but in many cases target data might not be available. So while complex correspondence detection does not require any instance data (except for enhanced strategies), but is less reliable, function detection is more reliable, but can only be performed if source and target instance files exist.

4.7 Using Patterns

4.7.1 Basics

The pattern algorithm is the only algorithm implemented in the complex correspondence and function detection module which exploits user-defined knowledge. It was therefore rather developed for research and several test cases, not as an outstanding additional feature.

The pattern algorithm offers two advantages:

- It may find new complex correspondences that other algorithms cannot detect.
- It may bring the elements using a concatenation function into the right order, without depending on instance data.

A pattern describes how a typical complex correspondence (like address or name) is normally structured and which elements (and element names) take part. Such a pattern consists of a unique name (id), a main term, and a set of subordinate terms, which is order-sensitive. Each term, including the main term, can have several synonyms.

For instance, the pattern 'Address' could have Address as its main term. Synonyms could be Addr, Place of Residence, Location, Adresse, etc. The subordinate terms of this pattern could be Street, No., Zip, City, Country. For each subordinate term there are several synonyms denoted, e.g., for Country the words State, Region, Land, etc.

The pattern therefore describes how a complex correspondence referring to an address should usually be designed. In the source schema, an element like Address, Addr, Place of Residence, etc. is to be expected, while in the target schema elements like Street, City, Zip and Country are expected. Using such patterns can yield new correspondences, or if correspondences have already been found, can put the sub-elements into the right order.

4.7.2 Working Principle

The pattern algorithm iterates all elements of the source schema and target schema, checking for each element whether it is either the main term of a pattern, or occurs as a subordinate term of the pattern (or as a synonym of one term). In either case, the pattern algorithm tries to find further nodes in the schemas which match the pattern. If it finds further nodes, the new correspondence will be created. If the concatenation function is necessary, the pattern algorithm will concatenate the elements in the right order, so, for instance, would change the original function

$concat(firstName,\ lastName,\ title)$ to $concat(title,\ firstName,\ lastName)$, that is, if the pattern requires this order. After implementing the order-detection algorithms as described before, this advantage of the pattern algorithm diminished; however, the pattern algorithm can still detect the correct order even without instance data.

4.7.3 Assessment and Prospect

The pattern algorithm works fine with frequent correspondences like the typical address and name function. It easily finds correspondences like ((*street*, *city*, *zip*, *country*), (*address*)) (note that we would not expect even one simple correspondence between those 5 elements), and the probability to generate totally false correspondences seems low, especially when the patterns are rather complex like in Address. However, if a schema does not use common formats, e.g., an address format like "Leipzig, Johannisgasse 26, 04109" (City, Street, No., Zip), the wrong order would be created. In this case, the pattern algorithm is even worse than the instance-based order detection.

To keep the probability of creating false correspondences low, the algorithm was extended by a mandatory attribute for each term. The main term is always mandatory, that is, it has to appear in the complex correspondence which is to be produced. However, for each sub-element exists a specification whether it is mandatory (so must occur in the schema) or not. For the address pattern, the sub-elements street, city and zip seem to be attributes which should be declared mandatory, whereas country and number could be declared as optional attributes, because they might not occur in every address correspondence. The algorithm creates the correspondence only if all mandatory sub-elements really occur in the schemas. If it finds an address element in schema A, and only the two elements Zip and $City$ in schema B, the correspondence would not be generated.

Some problems might arise if several elements in the schemas match a term, e.g., if there are several country-elements. False correspondences could be the result again.

Since the patterns are user-defined, it automatically fails at all other complex correspondences to which no pattern exists (and such correspondences normally dominate between large schemas). The pattern algorithm might be a possibility to detect complex correspondences which cannot be found by other algorithms, but needs much background-knowledge, and is therefore not a typical semi-automatic method anymore. Thus, the idea was not pursued any further, but might be another subject for future work.

4.8 Risks and Opportunities

Each function and complex correspondence generated by the program can be wrong, to begin with. The algorithms use rather intuitive and simple strategies to detect those functions, which would often work in little examples (standard cases), but fail in very complex, less-intuitive scenarios. We did not use established method likes machine learning or probabilistic reasoning, because they were too complex for the project, so we can hardly expect outstanding complex correspondence and function detection. The focus was always rather on comprehensible techniques and simple scenarios than on special cases. Yet apart from the correct functions that are detected, and which should be a great benefit for the user anyway, also the falsely detected functions and correspondences should not be completely despised. For every function and complex correspondence the program created there exists a true reason, based on rather sensible concepts, and even if they are not correct, the user might see some kind of relation between the two schemas, so creating the right function or correspondence could be much easier than without any function and complex correspondence detection. Besides, a completely wrong function can be detected by the user rather easily, and could be removed with one or two clicks, whereas detecting missing functions is a much more tedious process.

On the contrary, a common law is "Bad news travels fast". Producing too many wrong functions and complex correspondences could result in a worse user satisfaction than producing less, but more correct ones (smaller recall, higher precision). We therefore tried to keep the number of wrongly detected functions and complex correspondences as low as possible. The different thresholds implemented in the program, which could seem a little strict sometimes, might be a hint for this.

Also, the methods and ideas described in this chapter could be interesting for further research projects and COMA++ enhancements. There are still many opportunities to enhance the function and complex correspondence detector, so the ideas explained here might also serve as a solid base for future work.

4.9 Evaluation

4.9.1 Tests

Introduction

Testing and Debugging is always an important step in software engineering, and has to be carried out with much care and commitment in parallel to the development of software. For the implementation of functions in general, only few and simple test cases were necessary, however, the function detection and complex correspondence detection turned out to be a very complex undertaking, so many test cases had to be run. It was very crucial to test the many strategies to make sure that the desired complex correspondences are really detected (so that the program works as it was planned), and that, on the other hand, no pointless correspondences and functions are detected (which unfortunately is not always preventable). Furthermore, we had to make sure that there is no malicious effect on existing correspondences, e.g., that the program detects complex correspondences, but deletes other correspondences which were actually correct.

It was soon pretty clear that testing needed much regard, and could not be performed by simple debugging methods. Instead, we created a separate module with a full-fledged GUI to load, display and execute test cases. This debugging module was developed before even the first strategies were implemented.

Of course, there are many Java libraries that enable tests and debugging, especially the widely used JUnit, but since our program was so complex and had so many dependencies to other projects, we decided to create a separate test area where we could individually test all kind of mapping scenarios.

It was now possible to write test cases, and to load and to run them. This way it was easy to verify whether a strategy works as it was designed and, to a less degree, whether unintentional effects would occur.

Test Cases

Test cases are developed manually and have a specific format. Each test case consists of 3 mandatory files, and may have 2 additional, optional files. The three mandatory files are:

- A sources schema
- A target schema
- A list of (1:1)-correspondences (the "coma match result")

The source schema and target schema are the schemas for the map; of course we can assume that they always exist, because otherwise COMA++ could not have found any correspondences. The list of (1:1)-correspondences is that what we assume COMA++ would return when the program is integrated in COMA++ (remember that it was only integrated at the end, but we had to test it much earlier).

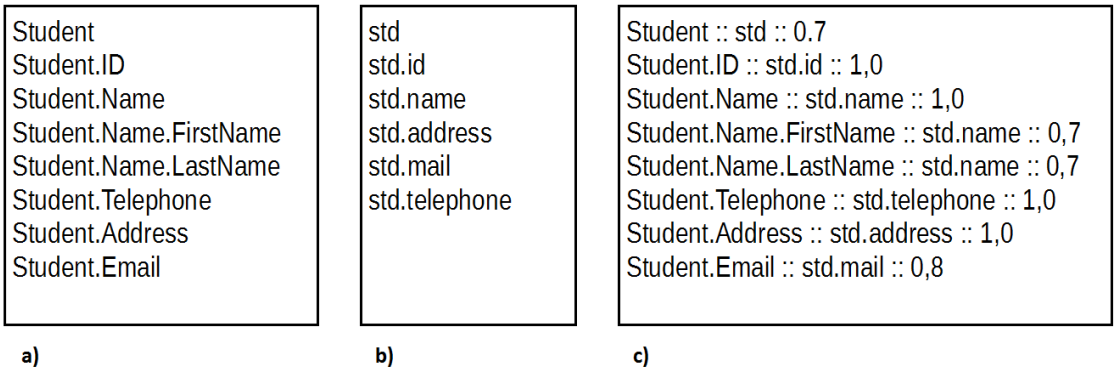


Figure 4.4: Sample test case to test the complex correspondence and function detection: **a)** The Source schema of the test case. **b)** The target schema of the test case. **c)** The supposed COMA++ match result. Executing this test case, the program should detect the complex correspondence ((FirstName, LastName), name) and remove all simple name correspondences.

Figure 4.4 shows how a simple test case could look like (that is, how the 3 files look like). Thus, a test case has a source schema, a target schema, and supposes that COMA++ has detected a list of (1:1)-correspondences.

There are two further files a test case may have:

- A source instance file
- A target instance file

These files are necessary to test some scenarios where instance data is important (e.g., to test the function detection or the order detection). In some cases, it is also recommendable to deliberately leave instance files out to look how strategies work if no instance data is available.

The GUI

The GUI consists of an output field to display the test cases and results, as well as various components to control the test cases. It is possible to display the source and target schema, to display both schemas at once, to display the coma match result, the instance data and, most importantly, the result produced by the program after a test case was executed. Furthermore, the protocols can be viewed, explaining why correspondences were created or not created.

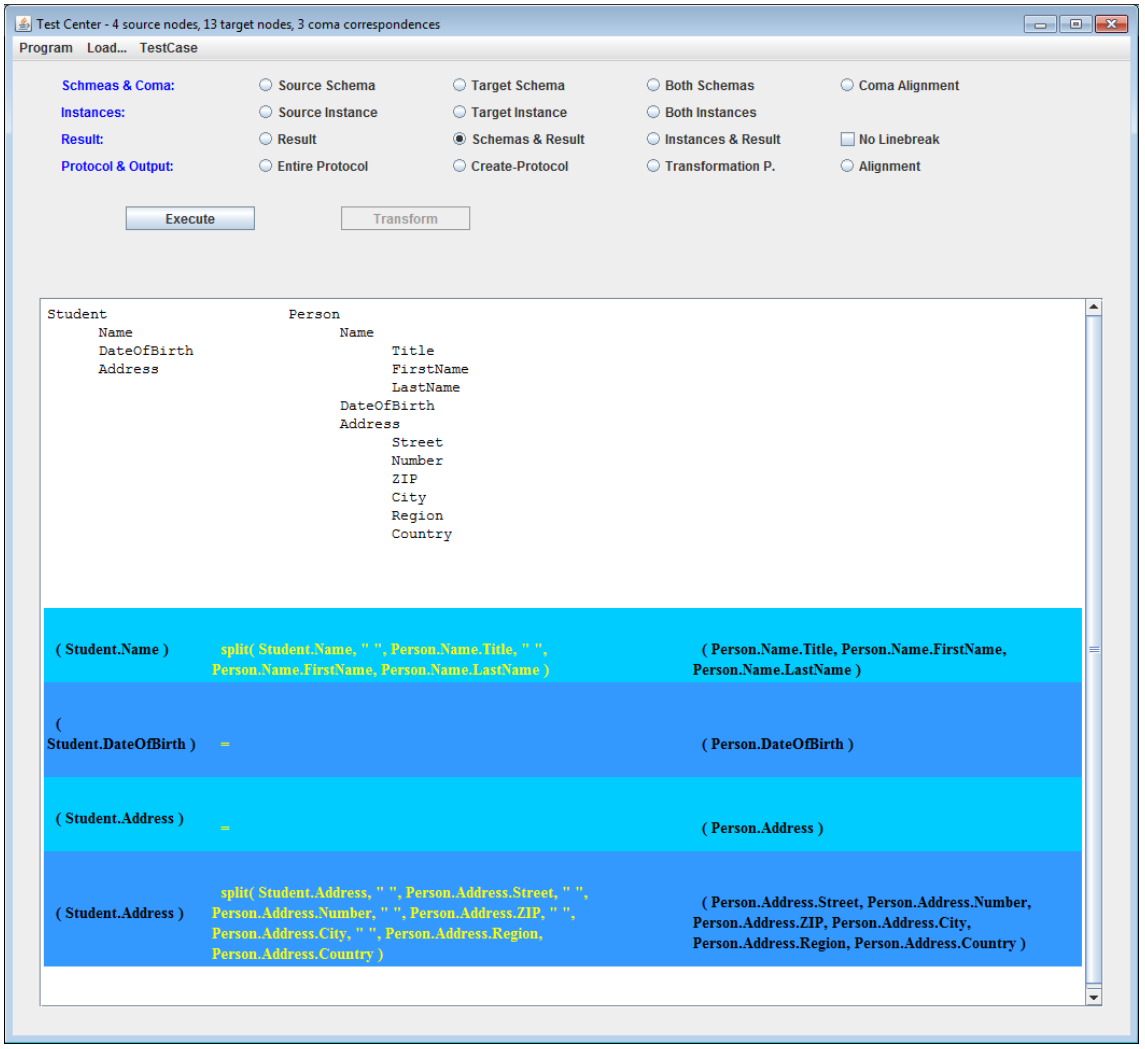


Figure 4.5: Screenshot of the program to test the complex correspondence and function detection.

There is also a transform button which allows to transform source data sets to the target file, so if instance data exists, the program can also be used to test data transformation. The data transformation uses a separate protocol where general occurrences, warnings and errors are logged. If a value cannot be transformed (e.g., if "Matthew Berger" is to be split into $(Title, FirstName, LastName)$, the program notes that this data object could not be transformed, and also gives the reasons why this error occurred; so even if the data transformation cannot always work smoothly, the user is able to see which objects could not be transformed.

To facilitate a more precise testing, it is also possible to enable and disable several detection strategies in the program. For instance, the pattern algorithm can be disabled, because it might effect the order detection of the complex correspondence detection strategies. With respect to the latter strategies, it is also possible to disable the scenario 2 or scenario 3 strategy (extended complex correspondence detection). In the final version of COMA++ this feature can be easily adopted, this way allowing

advanced users to disable the strategies which are less reliable (e.g., the replacement function).

Running Test Cases

Each test case concentrates on a little, rather specific scenario. For instance, there was a test case that checked whether a complex correspondence would be generated when a subelement has several subelements (so when we have to deal with sub-subelements). Another test case checked how the program would behave if a target node is connected with several source nodes. Only when these test cases were to our full satisfaction, the respective algorithm was considered finished, otherwise we revised it until it worked as intended.

The test cases were appropriate to test whether an algorithm works as desired, but not very appropriate to test whether it has some bad influence on other correspondences or even other algorithms. For this, the test cases were simply too little, and we had to look for larger scenarios to test the program – some we got from websites like www.stbenchmark.org.

4.9.2 Time Complexity

Introduction

Although the complexity of an algorithm can be exactly determined, the function and complex correspondence detection is far too complex, and consists of too many special cases to ascertain the exact time complexity. Basically, all (1:1)-correspondences detected by COMA++ are iterated in a loop, and for each correspondence a set of functions is tested, including the complex correspondence check. This would lead to $O(n^2)$, where n is the size of the schemas and the number of (1:1)-correspondences detected by COMA++. However, some functions exploit instance data, making it also necessary to walk through the instance files ($O(m)$, with m the number of instance entries); other strategies check whether there is another correspondence which has a target element connected with a different correspondence ($O(n^2)$).

At least, it can be said that the program has a polynomial time complexity; the exact complexity depends much on the respective schemas and constraints. If many special cases occur, and instance data is provided, a complexity of $O(n^5)$ or $O(n^6)$ could be easily reached.

Instead of focusing on the time complexity per se, we therefore decided to determine the execution time for different cases. Testing only a few different scenarios makes it already pretty clear what execution times we have to expect if two schemas are to be mapped.

The execution time was measured by the program itself, and is therefore rather precise. It depends on the hardware, as well as on the given schemas and coma match result. In fact, the execution time depends basically on 3 main parameters:

- The number of elements in the source and target schema.
- The number of (1:1)-correspondences detected by COMA++.
- The number of source and target instance data sets.

Increasing one parameter automatically increases the execution time. Actually one would have to distinguish between source nodes and target nodes (resp. source instance values and target instance values) again, but this would impair the execution time only little, so we decided to make it not any more complex; besides, it might be assumed that the number of source and target elements (resp. instance data) is normally rather similar.

Besides these parameters, the similarity between the two schemas plays a big part. If two schemas are very similar, the algorithm would soon detect many correspondences and finish quickly. If there are hardly any correspondences between the two schemas, the algorithm would seize on more specific methods which would make the process longer. Also the structure of the schemas (if it is more or less hierarchical) can influence the program immensely, especially the complex correspondence detection.

Last but not least, the execution time sinks if some features (some strategies) are disabled. However, when we determined the execution time, all features were regarded.

The Test Case Used for the Complexity Analysis

We created a basic test case having a source schema of 21 nodes and a target schema of 15 nodes (so altogether 36 nodes), as well as a pseudo COMA++ alignment of 10 correspondences. Both source and target schema are hierarchical; we tried to make them as natural as possible. To test larger scenarios, we simply reproduced the schemas and correspondences. Reproducing schemas and alignments 10 times brought us schemas of altogether 360 nodes and 100 correspondences, and later on 3600 nodes and 1000 correspondences. We even created schemas of 36,000 nodes and 10,000 correspondences for extreme examples, but in this case quickly reached the limits of our hardware.

Elements and Correspondences

First we tested the execution time depending on different numbers of elements and correspondences, i.e., we checked how the program behaves when the schemas become larger and the number of given (1:1)-correspondences increases. Instance data

Elements / Correspondences	10	100	1000	10000
36	1	2	130	11,800
360	4	10	910	94,500
3600	270	332	8,900	1015,000
36000	26,800	27,300	114,500	n/a

Table 4.6: Execution time in ms for schemas having different numbers of elements (rows) and correspondences (columns). In these scenarios only complex correspondences were discovered. The function detection was disabled, because no instance data was provided.

is neglected here, and since it is the premise of function detection, no function detection would be carried out in this scenario.

Table 4.6 reveals the execution time depending on the size of the schemas and the number of detected (1:1)-correspondences by COMA++. Note that most natural examples would be on the diagonal. A schema having 3600 elements but only 10 correspondences in between is as unlikely as a schema having 36 elements but 1000 correspondences in between. However, we tested all scenarios to show how the execution time behaves in general.

For small cases (36 nodes, 10 correspondences) and medium cases (360 nodes, 100 correspondences) the execution time is very low and the program terminates at once. If larger schemas are mapped, the execution time rises quickly. For large cases (3600 nodes, 1000 correspondences) almost 10 seconds will last until the program terminates. Still larger cases seem very difficult, because it might happen that the execution time would take many minutes or even hours, something which would not be acceptable for the user anymore. In most cases, schemas are not that large, though.

Regarding the table, there seems to be no real difference between the number of correspondences and the number of elements. A larger number of elements increases the execution time approximately in the same way as a larger number of correspondences would do. Thus, we came to the conclusion that the number of elements and correspondences increases the execution time rather in the same way. As we expected, the polynomial complexity leads to an immense rise of the execution time beyond medium scenarios.

The execution times seem very moderate both for the simple and the medium scenario. The large scenario is marginal, but still acceptable. Besides this, we estimated that the execution time of the COMA++ matchers is about 10 .. 100 times of the execution time of the complex correspondence and function module so that even larger execution times will not matter much.

Since the test cases we wrote ourselves were far from being real-word examples, we tested a few scenarios provided by STBenchmark as well. We already introduced STBenchmark in the third chapter, mentioning that it offers a selection of different

#Source Nodes	#Target Nodes	#(1:1)-Corresp.	Execution times
253	373	108	96 ms / 98 ms
463	633	357	182 ms / 175 ms
857	1122	105	119 ms / 124 ms

Table 4.7: Execution time in ms for scenarios generated by STBenchmark.

scenarios to judge the quality of a schema mapper, but STBenchmark also supplies a generator which produces random schemas to properly test a schema mapper. Although randomly generated, these schemas are much closer to real-world schemas than the ones we created manually. We therefore ran a few random scenarios and measured the execution time again. The results are presented in Tab. 4.7.

The first columns specify the size of the source schema, the size of the target schema and the number of correspondences detected by COMA++. This way, it becomes clear how large the scenario were that we tested. The fourth column expresses the execution time of the complex correspondence and function detector. The first value was the execution time for a mapping between source and target schema, and the second value for the other way around, so between target schema and source schema. As we had expected, and as it can be seen in the table, the execution time does not depend on the direction of the mapping, so mappings between source and target schema as well as target and source schema have always a similar execution time.

Elements and Instances

To analyze the function detector, it was necessary to offer instance data, and thus we created two additional instance files for the basic scenario, each consisting of 10 data sets for each schema. Since we had 36 nodes in the basic scenario, we had 360 single values in both instance files (resp. 3,600 for 360 nodes and 36,000 for 3,600 nodes).

To express the relation between the three parameters correspondences, nodes and instance sets, we combined elements and correspondences here. As already mentioned, this seems quite justified, because the number of elements and detected correspondences are normally interdependent. Thus, we regard 36 nodes having 10 correspondences in between, 360 nodes having 100 correspondences in between, etc.

As it can be seen in Table 4.8, the hardware even had trouble to process 36,000 instance values (1,000 data sets for both schemas). However, the number of instance data which is loaded for the function detection can be adjusted in the program. If an instance file exists that contains 10,000 data objects, it would be possible to load only 100 objects and work with this sample data to keep the execution time moderate.

As for the execution time, it seems rather similar to the previous analysis. The

execution time rises considerably beyond medium cases, and depends on the schema size and the number of instance data in the same way.

Elements (Corr.) / Instance Sets (Obj.)	360 (10)	3600 (100)	36000 (1000)
36 (10)	1.5	38	2,950
360 (100)	18	225	15,500
3600 (1000)	7,400	9,400	161,000
36000 (10000)	n/a	n/a	n/a

Table 4.8: Execution time in ms for schemas having different numbers of elements/-correspondences (rows) and instance sets/objects (columns).

4.9.3 Correctness

Introduction

There are different ways to judge the correctness and performance of an algorithm. The function and complex correspondence detection is a classic information retrieval problem so that determining recall and precision seems rather sensible. How many correct complex correspondences have been found (recall), and how many incorrect complex correspondences are among the detect ones (precision)?

To measure recall and precision is rather difficult. For a sophisticated analysis it would be necessary to map large schemas from external sources. Although we did this, it was hard to determine recall and precision formally, because mostly we were not able to completely understand the complex schemas, and could not decide which detected correspondences were actually true and which not. Additionally, checking so many produced correspondences would have been very tedious and expensive. We therefore rather used the large scenarios to test whether the program works in general, and to avoid some serious errors in special cases. All scenarios produced a proper map, and did not encounter serious errors. This way, we got convinced that the minimal goal, a terminating program, has been reached.

However, since the question about the correctness of the program was still not answered, we checked it manually to get at least an overview about the strengths and weaknesses of the function and complex correspondence detector. We tested little and medium scenarios, which had already been used to test COMA++ or ontologies, scenarios generated by STBenchmark, as well as scenarios created by ourselves (which we tried to make as natural as possible). It was a good way to find out the drawbacks and benefits of the program after all.

The problem of judging the effectiveness and correctness of the program was also caused by the large dependency to COMA++. Depending on the quality of the correspondences COMA++ detects, the algorithms produce more or less satisfiable results. If they get a set of imprecise or totally wrong correspondences, they cannot

produce exact ones. If they get precise and correct correspondences instead, they would be able to create more correct complex correspondences.

Complex Correspondences

We observed the following problems with regard to complex correspondence detection:

- Most incorrect complex correspondences were produced by the scenario 3 algorithms, i.e., the algorithms which consider correspondences on the same element level. We already stated that these correspondences are rather risky, and that if COMA++ produces different (1:1)-correspondences towards one element, where only one is correct, the program would create a false complex correspondence. Disabling the scenario 3 algorithms would avoid this problem, but some complex correspondences would not be found then – it is the classic recall-precision-problem.
- The number of incorrect complex correspondences in different element levels (scenario 1 and 2) was lower. If errors occurred, it was normally because COMA++ created a false or imprecise correspondence.
- We have generally problems with foreign keys in database schemas. It might occur that two foreign keys refer to an element, e.g., Novel.CategoryID and Movie.CategoryID refer to Category.CategoryID, and in this case the program would create a complex correspondence between those three elements. However, during the tests it turned out that COMA++ is able to deal with joins and foreign keys pretty well so that no wrong complex correspondences were created, because COMA++ did not generate the simple correspondences.

The result of these tests and analyses was that the dependency to COMA++ is larger than we originally assumed; since a schema matcher can never find all correct correspondences (resp. can never avoid false detections), we have to deal with supposedly incorrect data. Processing incorrect data normally does not result in correct data. However, we tested the program with a standard configuration of COMA++ – using other configurations could get a completely different result already. Besides, in some cases COMA++ works more precisely than the function and complex correspondence detector (as mentioned in the last item) so that it is also able to prevent incorrect decisions. From this point of view, the two programs cooperate pretty well after all.

STBenchmark Results

The problem of randomly created schemas is that they are quite different from real-word scenarios, and that it is hard to comprehend the correspondences. Nonetheless,

STBenchmark was a great help for us, because we could detect bugs and inconvenient behaviors that we maybe would not have detected by testing our little sample schemas.

Using the STBenchmark generator, we produced a map between a source schema having 166 nodes and a target schema having 194 nodes. COMA++ detected 104 correspondences, and our program discovered 6 complex correspondences. Analyzing this scenario, these 6 complex correspondences were correct, leading to a precision of 100 %. There were some further complex correspondences which the program would actually have detected, but since COMA++ did not discover the required (1:1)-correspondences, these complex correspondences were not generated. It turned out that these complex correspondences would have been wrong after all, because they were foreign keys, so as stated above, the precise COMA++ matchers often prevent the generation of a false complex correspondence.

Analyzing the schemas manually and scrutinizing the mapping file provided by STBenchmark, it seemed that there were no further complex correspondences between the schemas. This would mean a recall of 100 %, but it has to be acknowledged here that these values only refer to the one scenario we ran; for other scenarios they could be much lower or even 0. In retrospective, we also recognized that even the pretty large sample schemas by STBenchmark only cover a tiny part of all possible cases, and that they are not suitable to determine recall and precision in general.

Function detection

Testing the function detection was even more complicate, because we usually had no instance data for the test cases we ran. In the rare case that we had instance data, it was rather sample data so that there was no necessity for functions, and accordingly the program did not add any functions to the map.

For instance, to test the order detection, we would have needed two corresponding schemas where instance data is provided, and where an (1:n)-correspondence or (n:1)-correspondence exists, where the order of the subnodes is different from the order of the subvalues in the single node – as matter of fact, we could have spent many hours in searching for those scenarios without finding anything even close to these conditions.

We therefore tested the function detectors with test cases of our own, and for these test cases the function detectors worked fine and seemed stable. Except from the replacement function, all function detection strategies seem rather good, and a detected function is much more confident than a detected complex correspondence.

Chapter 5

Conclusions

In this Master's thesis we gave an introduction in functions and complex correspondences with regard to schema mapping, a rather specific but extensive and far-reaching field. We categorized and explained functions, we gave many examples to emphasize their importance for schema mappers, and we introduced studies where functions and complex correspondences were analyzed. Apart from the papers, we also presented a selection of schema mapping solutions, showing that there is no perfect program that is able to offer sophisticated schema matching, as well as a large set of functions, a clear user interface and ease of use. We specifically made clear that there is no program available yet that detects complex correspondences and functions automatically, although the iMAP dissertation is a very interesting and promising approach.

After the basics of functions and complex correspondences were described, we focused on the practical part, which was the implementation of functions in COMA++. We introduced the new data structure, which was necessary to represent functions and complex correspondences, and led over to the complex correspondence and function detection. This aspect was the most interesting part of the entire project, because this topic has hardly been considered so far, and we presented and explained many strategies to discover complex correspondences and functions within correspondences.

The tests showed that the complex correspondence and function detection does not always work perfectly, especially due to its strong dependency to the coma match result. Generally, the complex correspondence detection is more difficult and more prone to errors, but does not need instance data in its basic version. On the contrary, function detection seems less error-prone, but depends completely on source and target instance data, which might not always be available. At least we could show on many examples that a lot of common cases, like the classic name example, can be solved pretty well.

Throughout the entire practical part we explained the benefits and drawbacks of the implemented detection strategies, and we gave many notes how the strategies could be improved in future. Developing the detection methods was a new experience for us, and it has always been our goal to concentrate on rather intuitive and comprehensible strategies, instead of making the program more complicate than necessary. With this, many basic scenarios can be covered, even if the program fails in more difficult ones. Regarding this, the function and complex correspondence detector seems to be a valuable addition to the classic schema matcher COMA++.

There is still much space for improvement so that the solution we developed cannot be regarded as a complete or perfect one; we made this clear in the respective sections. However, the detection module seems already a very solid base, and it might be quite exciting to evolve it, because so many further aspects can be regarded and implemented in the new COMA++ module, which will now support the users in complex correspondence and function detection.

Bibliography

[1] Abteilung Datenbanken Leipzig. Coma++ — abteilung datenbanken leipzig. http://dbs.uni-leipzig.de/Research/coma.html, 2011.

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

[3] Altova, Inc. Altova mapforce - graphical data mapping, conversion, and integration tool. http://www.altova.com/mapforce.html, 2011.

[4] Robert Wilton Amor. A generalised framework for the design and construction of integrated design systems. Technical report, Ph. D. Thesis, Univ. of Auckland, New Zealand, 1997.

[5] Anhai Doan and Alon Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26:83–94, 2005.

[6] Dezernat für Öffentlichkeitsarbeit und Forschungsförderung Dr. Ulrike Pondorf. Universität leipzig, jahresbericht 2010. Technical report, Universität Leipzig, Germany, 2011.

[7] Elaine Chang. Heptox: Heterogeneous peer to peer xml databases. http://www.cs.ubc.ca/labs/db/heptox, 2005.

[8] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[9] Bin He, Kevin Chen-Chuan Chang, and Jiawei Han. Discovering complex matchings across web query interfaces: A correlation mining approach, 2004.

[10] IBM Corporation. Ibm - infosphere data architect - software. http://www-01.ibm.com/software/data/optim/data-architect, 2011.

[11] IBM Corporation. Ibm research — almaden research center — computer science: Schema mapping management system. http://www.almaden.ibm.com/cs/projects/criollo, 2011.

[12] Ningsheng Jian, Wei Hu, Gong Cheng, and Yuzhong Qu. Falcon-ao: Aligning ontologies with falcon. In *K-Cap 2005 Workshop on Integrating Ontologies. (2005)*, pages 87–93, 2005.

[13] Thomas Kudraß. *Taschenbuch Datenbanken*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 1. edition, 2007.

[14] M. A. Herná L. Pop. Mapping xml and relational schemas with clio. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 498–, Washington, DC, USA, 2002. IEEE Computer Society.

[15] Doan AnHai Belford Geneva; Winslett Marianne; Zhai ChengXiang Lee, Yoonkyong. Developing, tuning, and using schema matching systems. Technical report, Ph. D. Thesis, University of Illinois at Urbana-Champaign, 2010.

[16] Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, and Marcello Buoncristiano. Concise and expressive mappings with +spicy. *Proc. VLDB Endow.*, 2:1582–1585, August 2009.

[17] Microsoft Corporation. Biztalk server developer center. http://msdn.microsoft.com/en-us/biztalk/aa937640, 2011.

[18] MITRE Corporation. Openii. http://sites.google.com/site/openinformationintegration, 2011.

[19] Oracle Corporation. Data integration — fusion middleware — oracle. http://www.oracle.com/us/products/middleware/data-integration/index.html, 2011.

[20] Oracle Corporation. Mysql :: Mysql 5.5 reference manual. http://dev.mysql.com/doc#manual, 2011.

[21] SourceForge.net. Open information integration — download open information integration software for free at sourceforge.net. http://sourceforge.net/projects/openii, 2011.

[22] Arne Koschel Roland Tritsch Stefan Conrad, Wilhelm Hasselbring. *Enterprise Application Integration: Grundlagen - Konzepte - Entwurfsmuster - Praxisbeispiele*. Spektrum, 2006.

[23] Stylus Studio Corporate. Stylus studio - xml editor, xml data integration, xml tools, web services and xquery. http://www.stylusstudio.com, 2011.

[24] Tsinghua University, Beijing, P.R.China. Rimom. http://keg.cs.tsinghua.edu.cn/project/RiMOM, 2011.

[25] Frank Naumann Ulf Leser. *Informationsintegration*. dpunkt.verlag, 1. edition, 2007.

[26] Uwe Schneider und Dieter Werner. *Taschenbuch Informatik*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 6. edition, 2007.

[27] Universität Leipzig, Germany. Web data integration lab. http://wdilab.uni-leipzig.de, 2011.

[28] University Basilicata. The spicy project.
http://www.db.unibas.it/projects/spicy, 2011.

[29] University of Trento. Stbenchmark.
http://www.stbenchmark.org, 2009.

[30] Wikipedia, the free encyclopedia. Data modeling.
http://en.wikipedia.org/wiki/Data_modeling, 2010.

[31] Wikipedia, the free encyclopedia. Pareto principle.
http://en.wikipedia.org/wiki/Pareto_principle, 2010.

[32] Detlef Wille. *REPETITORIUM DER LINEAREN ALGEBRA, TEIL 1.* Binomi Verlag, 4. edition, 2003.

[33] Mikhail Zaslavskiy, Francis Bach, and Jean-Philippe Vert. Many-to-many graph matching: A continuous relaxation approach. In *ECML/PKDD (3)*, pages 515–530, 2010.

List of Figures

List of Tables

Listings