University of Alberta

A Framework for Semantically Verifying Schema Mappings for Data Exchange

by

Jagoda Katarzyna Walny

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2010
Edmonton, Alberta

Examining Committee

Dr. Denilson Barbosa, Computing Science

Dr. Marek Reformat, Electrical and Computer Engineering

Dr. Eleni Stroulia, Computing Science

Abstract

We propose a framework for semi-automatically verifying relational database schema mappings for data exchange.

Schema mappings for data exchange formally describe how to move data between a source and target database. State-of-the-art schema mapping tools propose several mappings, but require user intervention to determine their semantic correctness. For this, the user must understand the domain the schemas represent and the meanings of individual schema elements in relation to the domain.

Our framework eases the task of understanding the domain and schemas and performs preliminary mapping verification. We use a readable, expressive, and formal conceptual model - a domain ontology - to model the source and target schema domain. We model the schema semantics by *annotating* schema elements with ontology elements. Our mapping verification algorithm rewrites mappings as statements in terms of the ontology, and uses a reasoner to check that the statements are entailed by the ontology.

Acknowledgements

I would like to thank my supervisor, Denilson Barbosa, for his encouragement, enthusiasm, and direction. Thanks also to everyone who took the time to offer valuable insights, comments, and questions from a fresh perspective: our visitors Ashraf Aboulnaga, Paolo Atzeni, and Altigran da Silva, as well as members of the University of Alberta Database Group, particularly Paolo Cappellari and Filipe Mesquita.

Thanks to all of my friends in Edmonton for the great times I had while at the University of Alberta. In particular, thanks to Levi Lelis, Amir Ali Sharifi, Sapphire Zhao, and the rest of my office-mates for making ATH-155 a fun and memorable place to work; Andrew Neitsch and Jenn Gamble for the great lunches; and Ania and Andrzej Ottowicz, for welcoming me into their home.

Finally, thanks to my family and to Simon Nix, for your unending support and encouragement.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Symbols

Symbol	Description
S	source schema
T	target schema
$M : S \rightarrow T$	A multimapping of correspondences from S to T; the result of the schema matching step (step 1) in schema mapping creation.
$M_{exec} : S \rightarrow T$	A set of executable schema mappings from S to T; the result of the schema mapping generation step (step 2) in schema mapping creation.
$\mu : S \rightarrow T$	An executable schema mapping from S to T.
$q : q_T \leftarrow q_S$	A query in μ that maps elements of S to elements of T.
q_X	A conjunctive query over schema X
$r(\mathbf{a})$	A relation in a relational schema
$\mathbf{a}$	A list of attributes
a_i	An attribute at position i
$\mathcal{O}$	A domain ontology.
$\mathcal{C}$	A set of concepts in an ontology.
$\mathcal{P}$	A set of properties in an ontology.
$\mathcal{I}$	A set of individuals in an ontology.
α_r	A semantic annotation mapping for a relation r.
α^X	A semantic annotation for a schema X.
$\gamma_r(v, j)$	The concept associated with the variable v in relation r at position j.
$\gamma_r(v)$	The concept associated with the variable v in relation r.
$\Gamma_q(v)$	The concept associated with the variable v in a conjunctive query q.
$\delta_r(j)$	The key constraint set of position j in relation r.
$\Delta_r(v)$	The key constraint set of the variable v in relation r.

Chapter 1

Introduction

1.1 Motivation: Data Interoperability

Data interoperability occurs when an application can use data from one or more disparate data sources. With the amount of data being produced, stored, and exchanged in the world today, there are numerous situations for which achieving data interoperability is essential. For example:

- Multiple organizations with their own data storage schemas, such as regional government health services, might merge into one, larger organization and consolidate their data.
- A governing body may require its various organizations to submit annual performance data in a particular format; this format may change from year to year.
- Two separate organizations having data about a certain topic may wish to exchange or consolidate this data; however, they do not want to share private data about their employees and finances.
- A frequent traveler may wish to have a single interface that queries flight information from a number of airline and travel websites of their choice.
- A supplier may wish to exchange data with a manufacturer.

The common thread in the preceding examples is that the data to be exchanged and/or integrated comes from separate sources that were developed independently. This means that the data might reside in completely different formats - for example, some data might be stored in a relational database, the other as XML files. With the advance of Information Extraction, even textual sources can provide data. In addition, because each data schema is designed independently, these schemas will be different - even if they are expressed in the same data model (e.g. the relational data model) and describe the same domain. As stated by the authors of "The Lowell Database Research Self-Assessment" [Abiteboul et al., 2005]:

> "Any two schemas designed by different people will never be identical. They will have different units (one salary is in euros, another is in dollars), different semantic

interpretations (one salary is net including a lunch allowance, another is gross), and different names for the same thing (Samuel Clemens is in your database, but Mark Twain is in mine)."

Thus a central problem in achieving data interoperability is to find a way to overcome this semantic heterogeneity in schemas. Data exchange, data integration, and peer-to-peer approaches are a common way to do this.

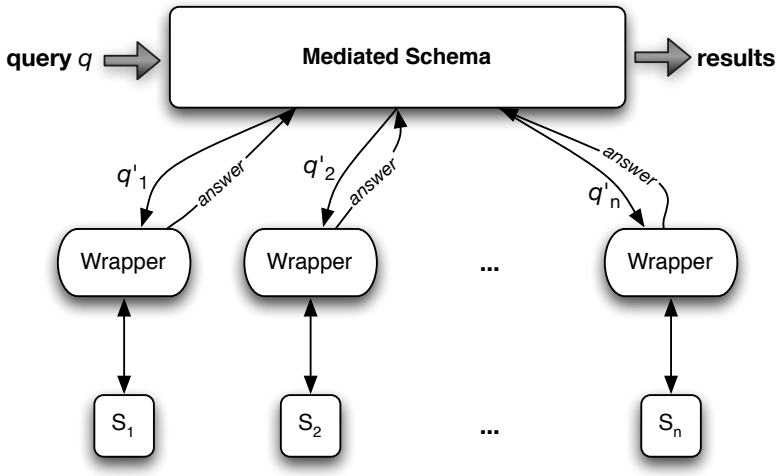

Figure 1.1: The mediated schema approach to data integration.

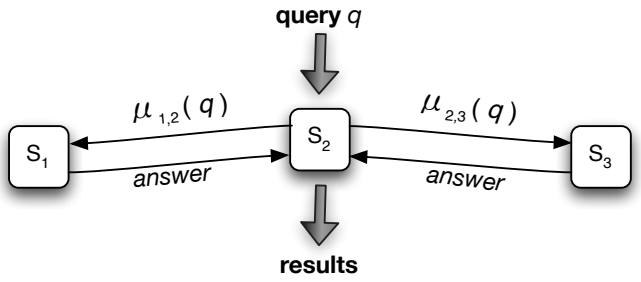

Figure 1.2: The peer-to-peer approach to data integration.

In data integration, a *mediated schema* is used to provide a uniform query interface for multiple data sources. The mediated schema approach is often used in enterprise data integration, for example when various branches of the same organization merge. In this approach, shown in Figure 1.1, the data stays in the individual source databases. Queries are expressed in terms of the mediated schema, while wrappers containing *schema mappings* between the source schemas and the mediated schema translate the queries and the results back and forth.

Another approach, often used in Web applications, is peer-to-peer data integration, as shown in Figure 1.2. In this approach, pairwise mappings are made directly between a number of individual

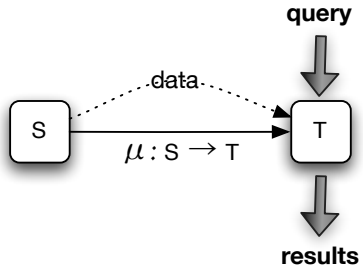

Figure 1.3: The data exchange approach to data integration.

database schemas, usually maintained independently at individual peers. Each peer has direct access only to its own schema and, through mappings, the data of its peers. There is no one unifying schema as in the mediated schema approach. Again, in this approach, the data stays in each individual source database and only the results of the queries are exchanged.

A third approach, data exchange, is shown in Figure 1.3. In this approach a mapping is created between a source and a target schema with the goal of moving all of the data from the source database to the target database. Queries are then made in terms of the target database.

The Use of Schema Mappings

As illustrated by the examples above, *schema mappings* are key to achieving data interoperability. A schema mapping is a precise specification of the relationships between the elements of a source schema and the elements of a target schema. This specification makes it possible to transform data from the source schema to fit into the target schema.

Executable schema mappings are schema mappings that can take an instance of a source schema and restructure it to meet the syntax and integrity constraints of a target schema. The source and target schemas need not be in the same format; for instance, the source database might be a relational database while the target database could be stored in XML. Executable schema mappings can be expressed in any executable language that can be used to extract data from or input data into the databases, such as SQL, XQuery, XSLT or PHP.

When creating schema mappings it is necessary not only to reconcile the syntax and integrity constraints of heterogeneous schemas; it is also important that the intended semantics of each schema is preserved. That is, it is important that elements in the source schema get mapped to elements in the target schema that represent the same real-world concepts.

In this thesis we focus on preserving semantics in schema mappings in the context of the data exchange approach to data integration.

1.2 Schema Mapping Creation

Given the importance of schema mappings and their potential complexity, a number of tools and techniques have been proposed to help in their creation. Generally, this process takes as input a source schema S and a target schema T, and outputs an executable mapping $\mu : S \rightarrow T$ which can be used to translate data from S to T. Deriving schema mappings is usually a three-step process, as shown in Figure 1.4. The three steps are: schema matching, mapping generation, and mapping verification.

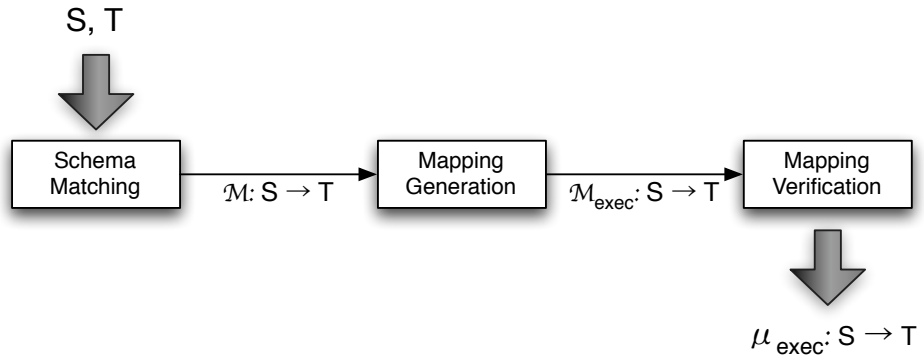

Figure 1.4: The schema mapping creation process

1.2.1 Schema Matching

The schema matching step involves finding correspondences between pairs of individual elements of the source and target schemas. (*Schema* matching is somewhat of a misnomer for this step; in fact, it is individual elements of schemas, such as attributes of relations, being matched - not the schemas themselves). Taking as input a source schema S and a target schema T, this step outputs a multimapping[1] $M : S \rightarrow T$, which consists of pairs of correspondences between elements of S and elements of T. (Where elements, in a relational database, are the attributes of relations).

Essentially, the schema matching step filters out those pairs of elements in the source and target schemas that are not likely to have a similar meaning. The methods used for this step use clues from the labels of the schema attributes [Cohen et al., 2003], the structures of the schemas [Melnik et al., 2002], and occasionally lexical comparisons to words present in external taxonomies of words [Pedersen et al., 2004]. The most effective schema matchers (e.g. LSD [Doan et al., 2001]) use a hybrid of these techniques. Even the best schema matchers do not achieve 100% accuracy - for example, Doan et al. [Doan et al., 2001] reported 71% - 92% accuracy for their hybrid matcher, LSD, and noted that two specific characteristics of schemas preventing the accuracy from being higher were: ambiguity in the meaning of labels, and being unable to anticipate every type of format

[1] *Multimapping* is a term used by Melnik et al. [Melnik et al., 2002] to describe a set of correspondences from which a number of potential mappings could be derived.

for the data. These deficiencies in accuracy are propagated to the next step in schema mapping creation, mapping generation.

1.2.2 Mapping Generation

Mapping generation tools (e.g. Clio [Miller et al., 2001b]) turn the pairwise correspondences from the previous step into executable mappings between the schemas; each mapping consists of executable queries from the source database to the target database. This step is not concerned with the semantics of the schemas - it is assumed that those are dealt with in steps one and three of schema mapping creation. The central problem in schema mapping generation is to create a mapping from the given correspondences, and to make sure that the mapping satisfies the specifications and constraints of the target schema, for instance: translating between heterogeneous schema types (e.g. relational to XML), ensuring foreign key constraints are met, or ensuring null values are used where appropriate, and ensuring that primary keys and foreign keys are not violated by the data that is mapped.

1.2.3 Mapping Verification

The mapping verification step involves verifying that the mapping generated in the previous step is accurate and reflects the intent of the source and target schema designers. In addition, where multiple alternative mappings are possible, a choice between them may be made. This step takes as input the executable mappings from the previous step and outputs an executable mapping $\mu : S \rightarrow T$ that meets the semantic constraints of the target schema as intended by the schema designer.

Some aspects of mapping verification are relatively simple - checking for datatype compatibility or primary key constraints, for instance. Such verifications involve violations against the formal schema specification, and are usually dealt with in the mapping generation step.

Other aspects of mapping verification, however, are more involved: for example, detecting subtle, semantic violations - those in which an element of the source schema is mapped to an element in the target schema with a compatible datatype but a different meaning. This is a highly understudied area - most schema mapping tools leave this part up to the user, with innovation being mainly at the graphical user interface level [Alexe et al., 2008], to make this step easier. This is not surprising, as such violations are impossible for a computer to detect without any extra information about the schemas. Users, on the other hand, are capable of gathering this knowledge elsewhere, for example by reading documentation, discussing with knowledgeable employees, or by using their own expertise and intuition.

For instance, see the example in Figure 1.5. Here two schemas, S and T, are shown, as well as some pairwise correspondences (a, b, c) between elements of the schemas. There is some inherent ambiguity present here that the schema matching step cannot deal with: the question is whether T`.Program.supervisor` refers to the technical leader of a project, or the man-

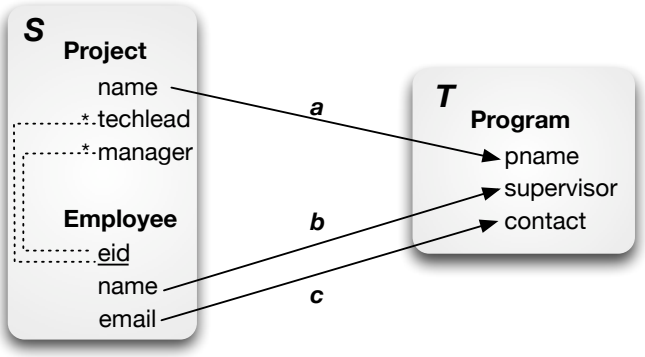

Figure 1.5: An example schema mapping verification problem

ager. Without any additional knowledge, it is impossible to algorithmically determine whether S.`Project.techlead` or S.`Project.manager` (or both) should be mapped to T.`Program.supervisor`. Even if we were to examine instances of data in the database, it is unlikely that the employee information about technical leaders (their names and emails) would have any unique characteristics that differentiate them from managers. In these kinds of cases, most schema mapping creation tools would at this point turn to the user to decide which mapping is correct (or which mapping is preferable, if more than one is correct).

This, however, is a significant problem. Such situations occur quite often in real-world schema mapping situations. Using commercial schema mapping suites, it still takes, on average, three man-months to complete a schema mapping [Sikka, 2006]. A considerable portion of this time is spent by the user on this mapping verification step. The user must spend time learning the meanings of individual elements in both schemas (which can be very large), then manually checking each mapping to see if it violates the intended semantics of the schemas.

1.2.4 Our Contribution

Mapping verification is the problem we study in this thesis. We approach the problem by exploring ways in which the schema designers' knowledge can be harnessed and formalized so that it can be taken advantage of computationally during the mapping verification step, reducing the amount of user effort required.

Our approach is formal and amenable to implementation in real systems. We envision it being used to enhance state-of-the-art schema mapping creation systems such as Clio [Miller et al., 2001b], replacing much of the work normally done by the schema mapping designer.

1.3 Illustrative Example

To further illustrate the schema mapping problem and the context of mapping verification, we present a more detailed example here.

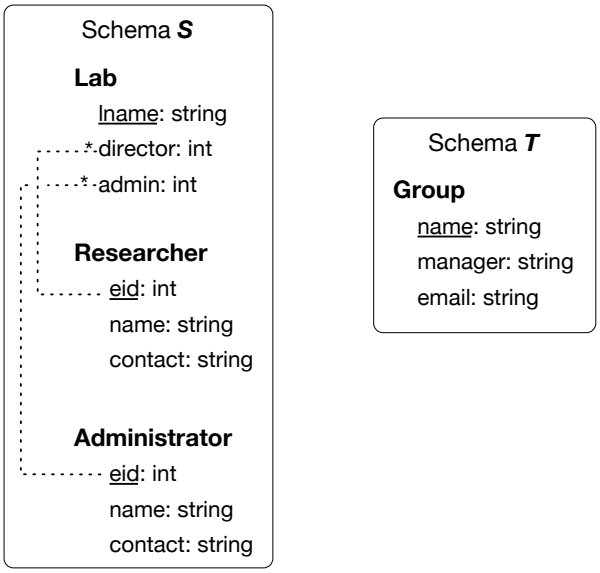

Figure 1.6: Schemas describing research labs in Company B (schema S) and the more general groups in Company A (schema T).

Suppose Company A is the parent company of a number of smaller companies. Each child company has a number of departments, which in turn have various groups working on different projects. Each child company also maintains its own databases. Company A decides it wants to keep its own database of information about each group in the child companies. Its information technology department creates a schema for this database and sends it to the child companies with instructions that each company must move the relevant data to Company A's database. Among other things, Company A is interested in keeping track of each group's name, manager, and the manager's email address. This schema subset is shown in Figure 1.6 as schema T.

Company B is a child company of Company A. Due to Company A's decision, Company B must find a way to move the required data to Company A's database. Company B assigns this task to Debbie, an experienced database expert at Company B.

Company B has a very large research department which is divided into a number of research labs. Company B's research labs correspond to Company A's "groups". The part of the schema that holds information about the research labs is shown in Figure 1.6 as schema S. Each research lab in Company B has a name, a lab administrator, and a lab director who is also a researcher. In this example we will demonstrate the process for finding a schema mapping from S to T.

Company B's schema S has three relations: `Lab`, `Researcher` and `Administrator`.

`Lab` describes a research lab in the company, with a primary key `lname` (the lab name), an attribute `director` (the researcher who directs the lab), and an attribute `admin` (the administrator of the lab). `Lab.director` is a foreign key to the `Researcher` relation, which describes employees of Company B who are researchers. `Researcher` has three attributes: the primary key `eid`, the researcher's name `name`, and the researchers's email address, `contact`. `Lab.admin` is a foreign key to the `Administrator` relation, which describes employees in administrative positions. Like `Researcher`, `Administrator` also contains a primary key `eid`, the employee's name `name`, and the employee's email address `contact`.

Table 1.1: An instance of Company B's schema S before data exchange occurs.

(a) `Lab`

lname	director	admin
BDataInterop	1	4
BRobots	2	3

(b) `Researcher`

eid	name	contact
1	Alice Smith	as@b.com
2	Bob Miller	bm@b.com

(c) `Administrator`

eid	name	contact
3	Carol Jones	cj@b.com
4	David Davis	dd@b.com

Table 1.1 shows an example instance of Company B's schema S before any data exchange occurs. This example will demonstrate how this instance is translated to fit with Company A's schema T.

Schema T describes the groups stored in Company A's database. It contains one relation, `Group`, which describes a group. `Group` has a primary key `name` (the group's name), an attribute `manager` containing the name of the group's manager, and an attribute `email` containing the email address of the group's manager.

Table 1.2: An instance of Company A's schema T before data exchange occurs.

(a) `Group`

name	manager	email
DMarketing	Ellen Allen	ea@c.com
Accounting	Filip Frost	ff@d.com

Table 1.2 shows an example instance of Company A's schema T before any data exchange occurs. This instance contains some data from other child companies of Company A.

Company B is faced with a data exchange problem: how can it translate the data stored in its database into a format that can be input into Company A's database? Furthermore, how can it

automate this data translation so that it can be repeated periodically?

One solution is to create an *executable schema mapping* between Company B's database schema and Company A's database schema. If the schemas to be translated are quite large, the executable schema mapping might be complicated to create by hand, so Company B would decide to use a schema mapping generation tool such as Clio [Miller et al., 2001b].

We illustrate how an executable schema mapping may be created to translate the project information from Company B's S to Company A's T using such a tool.

1.3.1 Schema Matching

Recall that the first step in schema mapping generation is to discover correspondences between schema elements in S and T, known as *schema matching*. A typical schema matcher might assign a similarity value to each pair of elements in $S \times T$, and set a threshold for the similarity value above which a pair of elements is considered likely to correspond. Suppose that our schema mapping generation tool finds the following correspondences for the database instances in Table 1.1 and Table 1.2:

(S.`Lab.lname`, T.`Group.name`, 0.98)
(S.`Researcher.name`, T.`Group.manager`, 0.90)
(S.`Administrator.name`, T.`Group.manager`, 0.89)
(S.`Researcher.contact`, T.`Group.email`, 0.92)
(S.`Administrator.contact`, T.`Group.email`, 0.93)

This indicates that `Lab.lname` and `Group.name` likely have the same meaning. However, it is difficult to determine whether `Group.manager` and `Group.email` refer to the name and email address of a lab director or administrator. `Researcher` and `Administrator` are very similar in their schema structure and there are no distinguishing characteristics in the instances of `Researcher` and `Administrator` - there is no inherent property of names and email addresses that would allow one to discern whether a particular name or email address belongs to a researcher or an administrator.

1.3.2 Generating Executable Schema Mappings

Our schema mapping generation tool uses the correspondences from the previous step to create executable schema mappings that can translate data from one schema to another. Clio does this in a two-step process. In the first step, it represents the two schemas according to a general model (this is to support mapping between schemas of different types); it then creates *logical mappings* by using the correspondences as well as the dependencies (foreign keys in relational databases) of the schemas. In the second step, it translates these logical mappings into the appropriate language for each schema (such as SQL, XSLT, or XQuery).

Clio could generate a number of mappings from the above correspondences; we present two possible logical mappings here, written in the syntax used by Clio. For more information about this syntax, see [Miller et al., 2001b].

The first mapping, μ_1, moves the contents of `Lab.lname` into `Group.name`. The mapping joins `Lab` and `Researcher` to move `Researcher.name` into `Group.manager` and `Researcher.contact` into `Group.email`.

μ_1:

for `Lab l, Researcher r` in S
where `l.director = r.eid`
exists `Group g in` T
where `g.name = l.lname` and
`g.manager = r.name` and
`g.email = r.contact`

The second mapping, μ_2, also moves the contents of `Lab.lname` into `Group.name`. However, it joins `Lab` and `Administrator` to move `Administrator.name` into `Group.manager` and `Administrator.contact` into `Group.email`.

μ_2:

for `Lab l, Administrator a` in S
where `l.admin = a.eid`
exists `Group g` in T
where `g.name = l.lname` and
`g.manager = a.name` and
`g.email = a.contact`

Essentially, μ_1 assumes that Company A would like to store the name of the lab director - the researcher - as the project manager, whereas μ_2 assumes that Company A prefers to store the lab's administrator as the manager.

Note that this mapping will alter the target database (Company A's database), but not the source, Company B's database.

A Note About Notation

For simplicity, in the remainder of this thesis, we write schema mappings as Datalog-style queries. In this notation, the mappings above would be written as:

$$\mu_1: \quad q_1 : T.\texttt{Group}(x_1, x_2, x_3) \quad \leftarrow \quad S.\texttt{Lab}(x_1, y, -), S.\texttt{Researcher}(y, x_2, x_3)$$

and

$$\mu_2: \quad q_1 : T.\texttt{Group}(x_1, x_2, x_3) \quad \leftarrow \quad S.\texttt{Lab}(x_1, -, z), S.\texttt{Administrator}(z, x_2, x_3)$$

1.3.3 Verifying Schema Mappings

We now have two mappings, both valid according to the constraints of schema T. We must now determine whether one or both of these mappings reflect the meaning of T as intended by its schema designer. This is the mapping verification step.

In this case, Clio will choose a small subset of the instance of S, apply the schema mappings, and show each mapping and the resulting instance of T to the user. The user then decides, based on

the results, the correctness of the mapping. Table 1.3 and Table 1.4 show the results of applying μ_1 and μ_2, respectively.

Table 1.3: An instance of Company A's schema T after data exchange if the manager is the lab director.

(a) `Group`

name	manager	email
AdCampaign	Ellen Allen	ea@c.com
Fin	Filip Frost	ff@d.com
BDataInterop	Alice Smith	as@b.com
BRobots	Bob Miller	bm@b.com

Table 1.4: An instance of Company A's schema T after data exchange if the manager is the lab administrator.

(a) `Group`

name	manager	email
AdCampaign	Ellen Allen	ea@c.com
Fin	Filip Frost	ff@d.com
BDataInterop	David Davis	dd@b.com
BRobots	Carol Jones	cj@b.com

Note that there is no discernible difference between the characteristics of the two instances. Syntactically, both are valid instances of T. Both have the same entries in `name`, both have employee names and emails that do not have any characteristics that distinguish them as being lab directors vs. managers. Neither instance has any information that would look out of place to a schema designer.

Now, consider the schema mapping designer, Debbie. She is an employee of Company B, and is quite knowledgeable about the meanings of the various database schema elements. She looks at the two suggested schema mappings, takes a moment to figure out how they are different, then realizes she is not actually sure which kind of manager Company A requires in their database - the one managing the research direction of the lab, or the one that deals with administrative issues? Or does Company A not have a preference? Debbie checks the documentation for T, but the description of the `manager` field states that it describes "the name of the manager of the group". Finally, she finds a contact at Company A who, after some discussion, decides that Company A would prefer to store the lab director's contact information, not the administrator's. So Debbie chooses μ_1 as the correct mapping.

Note that this small verification required background information that was not present nor apparent in the schema or instances of Company A's database. It also took quite a bit of time. If Debbie were only mapping the small subsets of the schemas that we see in this example, this would not be a problem. However, at a larger scale and with more complex mappings, this turns into a very time-consuming and error-prone part of the schema mapping creation task.

Although the schema matching step was able to guess the semantic correspondences for some of

the elements, there still remained some ambiguity that even Debbie herself was not able to resolve without turning to other sources. The approach presented in this thesis focuses on harnessing the knowledge that Debbie was able to obtain by looking at the schema documentation and discussing with Company A's schema designers.

1.4 Harnessing Knowledge

In order to harness the schema designers' knowledge, it must be captured in a formal conceptual model. For this, we find the Semantic Web research effort to be very useful.

The goal of the Semantic Web research effort is to create a World Wide Web that is more conducive to searching for complex information across multiple sources than the current Web is. In the current Web, information is annotated with instructions about how that information should be *presented* - for instance, which parts of the text should be shown in a bold or italic font, what size and colour the text should be, or where elements should be placed on the page and how they should be arranged. In the Semantic Web, information would also be annotated with its *meaning* in a machine-readable way, so that it would be much easier to automatically query across multiple data sources on the Web.

The goal of the Semantic Web research effort is to create a World Wide Web in which the meaning behind web page content can be understood by computers just as well as its structure, so that complex information can be found more easily. The Semantic Web community is an active research community.

Central to the data interoperability goal of the Semantic Web are *ontologies*, hierarchical descriptions of knowledge about a specific domain. They are derived from First Order Logic (FOL), and thus have precise interpretations and can be reasoned on. Ontologies are an established technology, with a number of standard representation languages (OWL is the standard ontology language for the Semantic Web), and available reasoners. They are frequently used as descriptions of data, both Semantic Web data and regular data in databases. Increasingly there is an industry forming around ontologies; they are already widely used in the biology research community for classification purposes.

We propose to use ontologies and formal reasoning to address the key problem in mapping verification: capturing the knowledge of the schema designers. An ontology can capture knowledge about the domain of the schema, in particular the properties of and relationships between elements. For example, the simple ontology represented graphically in Figure 1.7 describes the concepts Employee, TechStaff, and AdminStaff, and states that: every employee has a name and email; technical staff and administrative staff are both a type of employee (and thus also have a name and an email); and a technical staff member is not the same as an administrator.

In our framework, each schema is *semantically annotated* using concepts from the ontology, effectively enriching the schema with human- and machine-readable metadata in the form of rela-

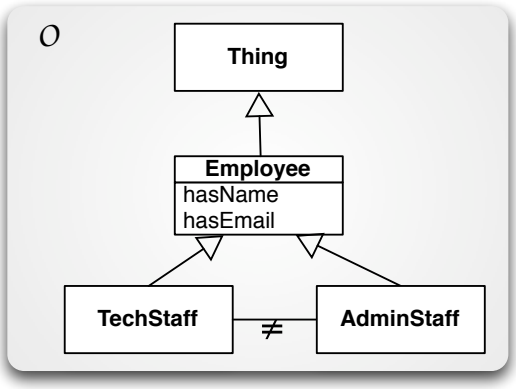

Figure 1.7: A graphical representation of a simple ontology.

tionships and properties. The ontology itself uses clear, unambiguous names for each concept, and describes a number of facts about the domain, making it an easier-to-understand conceptual model for this domain than a normalized schema with ambiguous or unclear attribute labels. We discuss ontologies in more depth in Chapter 3.

1.5 Overview

The remainder of this thesis is organized as follows: Chapter 2 describes relevant work in the field of data interoperability, as well as work describing ontologies, reasoning, and data interoperability using ontologies. Chapter 3 explains ontologies and reasoning in more detail.

Chapter 4 formally defines the mapping verification problem we address in this thesis. Chapter 5 introduces our framework and describes an algorithm for finding concepts in an ontology associated with schema elements as well as an algorithm for performing mapping verification using ontologies and reasoning.

Chapter 6 and Chapter 7 provide a discussion of our proposed framework and describe our vision for future applications of this framework.

Chapter 2

Related Work

The work presented in this thesis brings together two separate, but related disciplines: data exchange between heterogeneous data sources and the Semantic Web. In this chapter we outline work related to both disciplines.

2.1 Part I: Managing Heterogeneous Data Sources

Facilitating interoperability of heterogeneous data sources is an important problem in the field of data management. Applications use various data formats and schemas; in order for these applications to communicate with one another, they must be able to understand each others' data.

Two main approaches to data interoperability are *data integration* and *data exchange*. Data integration focuses on allowing queries to be answered over a number of data sources; this is done using either mediated schemas or views. Data exchange involves transforming an instance of a source schema into an instance of a target schema. Both data integration and data exchange rely on schema mappings.

There are two main types of data interoperability settings, which we call *critical* settings and *casual* ones.

Casual data interoperability settings involve peers exchanging small amounts of data in situations where errors and inaccuracies can be tolerated; for instance, users mashing up data on the Web using a tool like Potluck [Huynh et al., 2008], or two colleagues integrating their calendars. Here, failure to map the data correctly is not catastrophic.

Critical data interoperability settings occur in business, health care, or similar settings. Typically, they use very large schemas, are expensive, and might take a few months to complete. Those involved in critical data integration get paid to do it and make use of existing schema integration tools such as Clio [Miller et al., 2001b] to find the schema mappings. In addition, they may use the same schema in more than one data integration project over the long-term. Here, accuracy is very important, thus user involvement in the process is necessary; failure to map the data correctly can be disastrous.

In this thesis we focus on the critical data interoperability setting.

2.1.1 Data Integration

Data integration systems provide users with a single query interface for multiple heterogeneous data sources. The basic approach to this problem is to reformulate queries that the user makes over a mediated schema in terms of a number of source schemas.

One approach is to have a single, global mediated schema with separate mappings to each source schema. The earliest approaches [Batini et al., 1986] defined the global schema as a view over the source schemas, known as the Global-As-View (GAV) approach. The Information Manifold [Levy et al., 1996] introduced the Local-As-View (LAV) approach, in which each data source is described as a view over the mediated schema; this approach made it easier to add new source schemas after the mediated schema was created. Lenzerini [Lenzerini, 2002] compares the LAV and GAV approaches from a theoretical perspective. LAV and GAV can be combined to form the Global-Local-As-View (GLAV) approach [Friedman et al., 1999].

In other approaches, multiple mediated schemas are used. TSIMMIS [Garcia-Molina et al., 1995] allows the integration of various sources (including semi-structured sources written in XML) using a number of mediated schemas as well as translators between data sources and mediated schemas. In peer-to-peer data sharing (e.g. [Kementsietsidis et al., 2003]), peers in a network maintain mappings between their own schemas and those of their neighbours, but there is no one global schema across all sources.

An overview of recent progress in the field can be found in "Data Integration: The Teenage Years" [Halevy et al., 2006].

2.1.2 Data Exchange

The framework presented in this thesis is based upon data exchange. The goal of data exchange is to move data structured under a source schema to a database structured under a different target schema. This approach is well-suited to a scenario where two groups merge and wish to consolidate their data in one database, under one schema. The data exchange problem was formalized by Fagin et al. [Fagin et al., 2005a]; they provide a theoretical discussion of the data exchange problem in [Fagin et al., 2005b].

In practise, data exchange is executed using extraction, transformation, load (ETL) tools, usually commercial ones such as Microsoft's SSIS [Wyatt et al., 2009]. ETL tools extract data from heterogeneous data sources, transform it to fit a target database (this includes checking integrity constraints and ensuring primary and foreign key relationships are preserved), and load it into the target database. These tools focus on solving performance issues - how to extract, transform, and load large amounts of data in the shortest amount of time once the source and destination of the data is known.

In contrast, we are more concerned with helping the user - the database expert involved in the data exchange - to determine which parts of the schemas from the source databases correspond to which parts of the schema of the target database. Once this information is known, it can be input into an ETL tool so that the actual data exchange is performed.

2.1.3 Designing Schema Mappings

In order to exchange data from a source to a target database, it is necessary to find an executable mapping from the source to the target schema that satisfies both the formal and the semantic constraints of the target schema. Approaches to finding schema mappings can generally be divided into three steps: schema matching, schema mapping generation, and schema mapping verification. (Some tools perform more than one step at once).

Schema Matching

Schema matching involves finding correspondences[1] between individual elements of the source and target schemas. For relational databases, these are generally correspondences between attributes of relations rather than the relations themselves.

Schema matching techniques build on a variety of similarity metrics, both for atomic (e.g. attribute names) and composite (e.g. relations) schema elements. Similarity metrics use structural, lexical, statistical, and semantic characteristics of the schemas and their elements to quantify the semantic distance between schema elements.

The most basic similarity metrics are *lexical*: they examine the structures of words and sentences, for example string-distance metrics [Cohen et al., 2003]. *Structural* similarity metrics examine the placement of elements inside the schema structure. For example, Similarity Flooding [Melnik et al., 2002] represents schemas as graphs, then uses both lexical and structural similarities between nodes in the graphs to choose correspondences between schema elements.

A number of similarity measures attempt to quantify the semantic distance between elements using information beyond what is contained in the schemas themselves. Some methods, such as multi-column substring matching [Warren and Tompa, 2006], rely on comparing instances of data in the source and target database to determine which elements or groups of elements correspond to each other. A number of semantic similarity measures, such as WordNet::Similarity [Pedersen et al., 2004] and some information theoretic definitions of similarity [Lin, 1998] are based on WordNet [Fellbaum, 1998], a taxonomy that groups English words into sets of synonyms called *synsets*. These synsets are organized in a hierarchy based on their hypernyms and hyponyms. Note that for these approaches it is necessary to have labels that appear in the WordNet dictionary - mainly English words. Other text corpora may be used instead of, or in conjunction with, WordNet [Li et al., 2003].

[1] Note that, in the literature, the terms "matches" and "mappings" are often used interchangeably to refer to "correspondences", the result of the first step in finding schema mappings. Throughout this thesis, we use "matches" to refer to correspondences, and "mappings" to refer to executable mappings - the result of the second and third steps of the schema mapping creation process.

Schema matching is a very difficult problem [Gal, 2006]. To maximize precision, most schema matching tools use a variety of similarity metrics, some designed specifically for matching schemas, others more general-purpose. Some prominent schema matching tools that use this hybrid approach are COMA++ [Aumueller et al., 2005], Cupid [Madhavan et al., 2001], and LSD [Doan et al., 2001, Doan, 2002]. For a comprehensive survey of schema matching approaches, see "A survey of approaches to automatic schema matching" [Rahm and Bernstein, 2001].

Schema Mapping Generation

The second step of finding schema mappings is the generation of executable mappings that can move data from the source to the target schema. These executable mappings must match all of the integrity constraints of the target schema. Many mapping generation tools also include their own schema matching tools.

Mapping generation tools are available for a variety of schema types. Clio [Miller et al., 2001b, Popa et al., 2002] can create correspondences and executable mappings between schemas in various languages (such as XML and relational schemas). FleDEx [Mesquita et al., 2007] is a framework for exchanging data in semi-structured data sources; its Data Fitting method both matches attributes and creates the executable mapping between schemas.

Schema Mapping Verification

The third, and final, step in creating schema mappings is to verify the semantic correctness of these mappings. That is, to ensure that when data is mapped from an element s in a source schema to an element t in a target schema, that s and t represent the same real-world concepts. This is done by examining the executable mappings produced in the mapping generation step. The end goal is to have a mapping that is not only syntactically correct (as ensured in the mapping generation step), but also *semantically* correct.

Due to the inherent difficulty of this task - it requires a knowledge and understanding of the *meanings* of schema elements in both schemas - it is difficult to automate; this is the least-explored step in schema mapping creation. The majority of current schema mapping tools defer the task of verifying schema semantics to the user, which can be a time-consuming process when dealing with large schemas.

However, there have been several attempts to partially automate this task. Rull et al. [Rull et al., 2008] validate schema mappings by examining a number of properties of these mappings, including query satisfiability and number of null values generated. MUSE [Alexe et al., 2008] shows the user examples of data instances translated by the schema mapping to preview the effects of the mapping on the data; this approach still requires a high level of user involvement. The Spicy approach [Bonifati et al., 2008] compares the data instances translated by the schema mapping from the source to the data instances originally in the target; this approach is dependent upon the quality

of the original target database instance and its inherent similarity to the source data to be mapped, as well as the existence of detectable correlations between data in the source and target instances.

2.1.4 Model Management

A more abstract approach to data interoperability is model management. The goal of model management [Bernstein and Melnik, 2007] is to create generic tools for matching, merging, and translating schemas created in terms of various heterogeneous data models. These tools involve operations on specific *types* of schemas (e.g. relational schemas, XML schemas), rather than particular instances of schema types.

2.2 Part II: The Semantic Web

2.2.1 The Semantic Web

A significant amount of research is being performed with the goal of making the Semantic Web vision a reality. In this vision, the Semantic Web is an extension of the World Wide Web in which all publicly available data is annotated so that any independently designed data source is understandable to both machines and humans. This would allow the creation of applications that can tap into the vast data stores on the Web regardless of when or by whom the applications were created. Today's Web is highly focused on displaying data, whereas the Semantic Web is focused on easily integrating and reusing data of any form (including textual data displayed on Web sites). The creation of a Semantic Web is a substantial, ongoing research effort that draws on knowledge from many disciplines, including databases, knowledge representation and reasoning, information retrieval, machine learning, natural language processing, peer-to-peer networks, Web services, and more. For a thorough overview of this research, see *The Semantic Web* [Kashyap et al., 2008]. For an overview of established Semantic Web standards, see the W3C's Semantic Web website[2] or the official Semantic Web wiki[3].

Semantic Annotation of Data

The backbone of the Semantic Web is annotation. Rather than having a common annotation scheme, data owners can design their own annotation scheme and specify the semantics and vocabulary of the annotation scheme in a formal, machine-readable manner using an *ontology*. While there is no single, universally agreed-upon definition of an ontology, [Noy, 2004] provides the following summary of the most common definitions:

> "...the common thread in these definitions is that an ontology is some formal description of a domain of discourse, intended for sharing among different applications, and expressed in a language that can be used for reasoning."

[2] http://www.w3.org/2001/sw/
[3] http://semanticweb.org/

We describe ontologies based on Description Logics, which we use in this thesis, in more detail in Chapter 3.

Ontologies are represented using a well-defined knowledge representation language, which determines its expressiveness and the complexity of the reasoning which can be performed on the ontology. Ontology representation languages are derived from description logic (DL), which consists of decidable subsets of first-order logic (FOL). For an overview of knowledge representation, see "What is a Knowledge Representation?" [Davis et al., 1993]. For a thorough overview of DL, see *The Description Logic Handbook* [Baader et al., 2007]. For a description of the use of DL as ontology languages, see [Baader et al., 2005]. The standard ontology language for the Semantic Web is the Web Ontology Language (OWL), which comes in three versions, each of varying expressiveness: OWL Lite, OWL DL, and OWL Full. In our framework we use OWL DL, a language equivalent to the description logic language $\mathcal{SHOIN}(\mathbf{D})$ [Horrocks and Patel-Schneider, 2004]. For the full specification of OWL, see the W3C OWL Language Reference[4].

Reasoning

A fundamental aspect of using ontologies is reasoning on them, that is, inferring new facts about the ontology and verifying the consistency of statements in the ontology. Although reasoning on FOL is not decidable in general, there are subsets of FOL (such as OWL DL), on which reasoning is decidable. For a thorough background to reasoning on knowledge representation languages, see *Knowledge Representation and Reasoning* [Brachman and Levesque, 2004].

Reasoners exists for various knowledge representation languages. For some examples of reasoners directly related to FOL see [Patel-Schneider, 1990] and [Tsarkov et al., 2004]. Reasoners for OWL DL include RACER [Haarslev and Möller, 2003], Pellet [Parsia and Sirin, 2004], and FACT++ [Tsarkov and Horrocks, 2006].

2.2.2 Data Interoperability on the Semantic Web

There is substantial research both in data interoperability using ontologies and in *ontology alignment*, the integration of ontologies. For a discussion of the importance of ontology alignment for interoperability on the Semantic Web, see [Shahri et al., 2008].

Ontology Alignment

Ontology alignment is the integration or merging of two or more ontologies. There are several parallels between aligning ontologies and finding schema mappings. Many ontology alignment approaches use lexical and structural similarity measures. For example, [Ehrig and Sure, 2004] use neural networks to combine various similarity measures for ontology alignment, and find that this hybrid approach is superior to using a single similarity measure. A machine learning approach to

[4] http://www.w3.org/TR/owl-ref/

ontology alignment is GLUE [Doan et al., 2003, 2002], which classifies instances of concepts in one ontology using a classifier trained on instances of concepts in another ontology to find overlapping concepts. Some ontology alignment approaches [Qin et al., 2007, Lei, 2005] focus on creating executable semantic mappings between ontologies. Zhdanova and Shvaiko [Zhdanova and Shvaiko, 2006] propose a community-driven approach to ontology alignment, so that distributed groups can both contribute to and reuse mappings between ontologies.

Where ontologies differ from relational schemas is in the possibility of using reasoning to determine semantic compatibility between concepts in the ontologies. (This ability is fundamental to our mapping verification framework.) Bouquet et al. [Bouquet et al., 2003] approach ontology alignment from a logical perspective, framing it as a problem of satisfying logical formulae derived from the ontologies rather than relying on lexical or structural similarities between the ontologies. They find that this approach performs better than those that rely on lexical and structural similarities alone. Meilicke et al. [Meilicke et al., 2006] and Udrea et al. [Udrea et al., 2007] *debug* ontology mappings by using reasoning to verify their satisfiability, an approach that inspired our own.

For an overview of the state-of-the-art in ontology alignment, see Noy [Noy, 2004], Kalfoglou and Schorlemmer [Kalfoglou and Schorlemmer, 2003], or Choi et al [Choi et al., 2006].

Ontologies and Data Interoperability

Ontologies have been used in various ways to facilitate data interoperability. Doan and Halevy [Doan and Halevy, 2004] have written an overview of semantic integration research in the database community. Aleksovski et al. [Aleksovski et al., 2006] match unstructured vocabulary lists to each other using an ontology as background knowledge. Embley et al. [Embley et al., 2004, Embley, 2004] use snippets of ontologies in a schema matching tool. Calvanese and DeGiacomo [Calvanese and De Giacomo, 2005] align a federated database schema to an ontology so that queries against the databases are made in a language more natural to the user. Giunchiglia and Shvaiko [Giunchiglia and Shvaiko, 2004] turn schemas into ontology-like hierarchies and reason on them to facilitate schema matching.

There has also been some work on linking relational databases with ontologies. MapOnto [An et al., 2005, 2006] semi-automatically annotates relational database schemas with semantic information from an ontology. Poggi et al. [Poggi et al., 2008] present a new ontology language and a language for mapping between database schemas and ontologies. Motik et al. [Motik et al., 2007] extend OWL with integrity constraints.

2.3 Conclusion

We have presented an overview of data interoperability issues related to both data integration and data exchange, particularly in the realm of creating schema mapping for data exchange and verifying these mappings. We also introduced some Semantic Web technologies (ontologies and reasoning)

intended to facilitate data interoperability on the Web and noted some parallels between traditional data interoperability issues (finding schema mappings) and newer interoperability issues on the Semantic Web (aligning ontologies). We noted that in ontology alignment, approaches that take advantage of the ability to reason perform better than those that rely only on structural and lexical properties; this ability is being integrated into traditional approaches for data interoperability, as in this thesis.

In the next chapter, we examine ontologies and reasoning in more detail.

Chapter 3

Ontologies, OWL, and Reasoning

Ontologies are conceptual models that describe some domain of knowledge about the world, often referred to as the *domain of discourse* or the *universe of discourse*, or simply *domain*. Recall Noy's summary of the most common definitions of an ontology [Noy, 2004]:

> "...the common thread in these definitions is that an ontology is some formal description of a domain of discourse, intended for sharing among different applications, and expressed in a language that can be used for reasoning."

We consider an ontology to be a *classification* of real-world concepts into a hierarchical taxonomy or a graph of concepts, their properties, and the relationships between them. Essentially, it is a *knowledge base*, consisting of a model - a terminology for the domain - and individual instances of this model.

A key advantage of ontologies as compared to other types of conceptual models (such as UML or database schemas) is their expressiveness, structure, and readability. They are easy to understand because they support abstraction, encourage descriptive concept names, and are expressive enough to allow a large number of details about the domain of discourse to be described.

Ontologies are described using languages that support *reasoning*. That is, an ontology can be input into a reasoner, which will check that the ontology contains no contradictory statements and will perform automatic classification. Automatic classification uses the *facts* - statements inside the ontology relating to concepts, properties, and relationships - to derive new facts that may not have been evident to the ontology designer.

Ontologies are a general construct that can be applied in a variety of situations. Among previous applications of ontologies are: as knowledge bases for intelligent agents in artificial intelligence; as taxonomies of biological terminology; and as shared vocabularies in Semantic Web applications. In addition, there are various types of ontologies. Some, like the Suggested Upper Merged Ontology (SUMO) [Niles and Pease, 2001], aim to broadly define terms that other ontologies may use. Others describe small parts of very specific domains.

Due to the varying needs of each application, many ontology representation languages have been proposed over the years, each with varying degrees of expressiveness and decidability. These have included, notably, the Resource Description Framework (RDF)[1], DAML+OIL [Horrocks, 2007], and the Web Ontology Language (OWL)[2].

A very expressive language for representing knowledge is First-Order Logic (FOL), but reasoning on FOL is not tractable. Thus, a class of languages containing decidable fragments of FOL has been created; these languages are called Description Logic, or DL, languages.

We have chosen to use one variant of OWL, known as OWL-DL, as the basis for our framework and the discussion in this chapter. OWL-DL is known to be equivalent to the Description Logic language $\mathcal{SHOIN}(\mathbf{D})$. OWL-DL has an appropriate level of expressiveness for our purposes, yet is still practical in terms of reasoning costs.

This chapter outlines the basic concepts needed to understand ontology modeling in OWL-DL as well as the basics of reasoning on ontologies. We do not discuss OWL syntax, but we do explain the relevant properties and limitations of OWL-DL. To learn about OWL syntax, see Chapter 4 of *A Semantic Web Primer* [Antoniou and van Harmelen, 2004]. Throughout this chapter and this thesis we use standard DL syntax. For more information about this syntax, see Appendix 1 of *The Description Logic Handbook* [Baader et al., 2007].

3.1 OWL

The Web Ontology Language (OWL) is the standard language for modeling ontologies on the Semantic Web. There are three versions, each with differing expressiveness. OWL Lite and OWL-DL are both based on description logics (OWL-DL is more expressive). Both are decidable languages for reasoning. OWL Full is much more expressive, but not decidable. We use OWL-DL.

3.2 Describing Ontologies in OWL-DL

There are two main parts of an ontology: the *TBox* and the *ABox*. The *TBox* describes the *terminology* of the domain, what is known as *intentional* knowledge. It is what describes the general structure and relationships of the domain. The *ABox* contains *assertions* about the domain, what is known as *extensional* knowledge. It describes an individual instance of the domain. The relationship between a *TBox* and and *ABox* is, in some ways, similar to the relationship between a relational database schema and an instance of that database.

There are three main components in an ontology: classes, properties, and individuals. Definitions of classes and properties make up the *TBox*; definitions of individuals make up the *ABox*. Classes are sets of individuals; properties are binary relationships between classes and other classes

[1] http://www.w3.org/RDF/
[2] http://www.w3.org/TR/owl-ref/

(known as *object properties*) or between classes and values (known as *data properties*). Properties can also be applied to individuals.

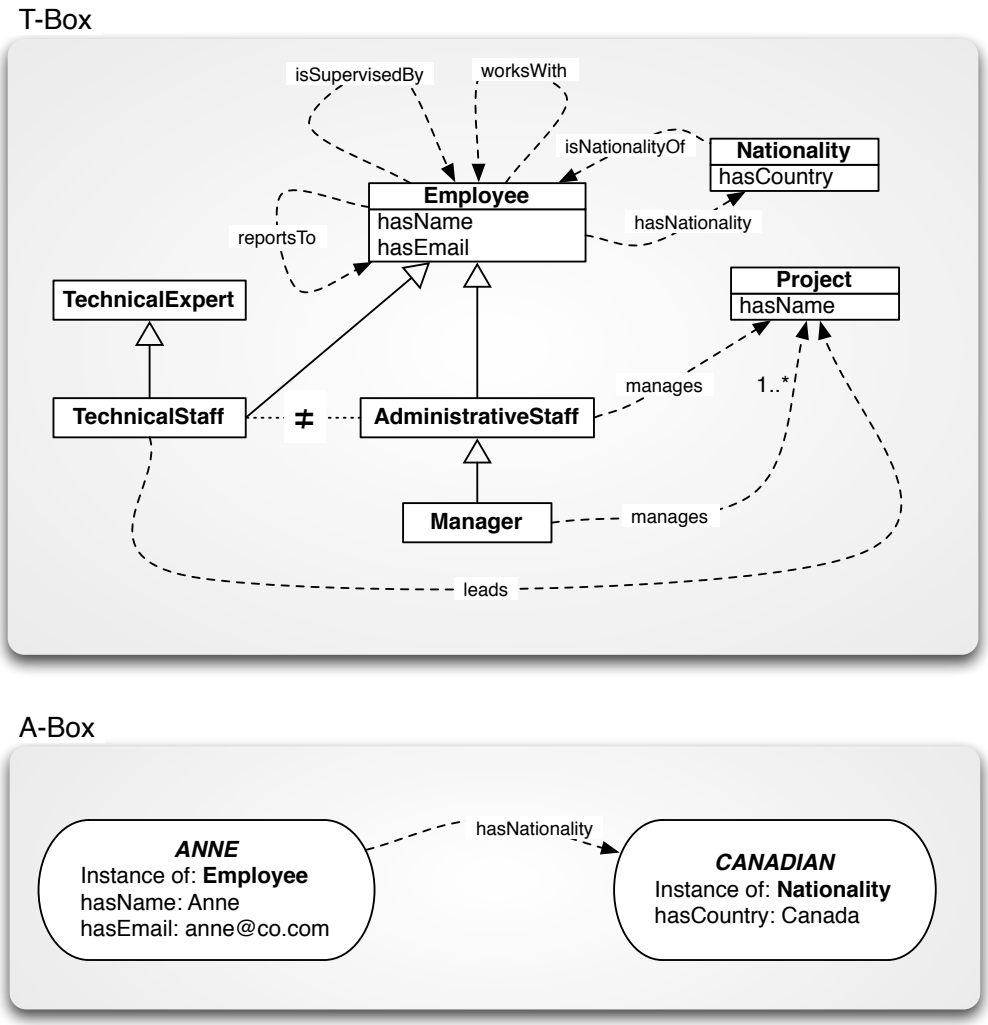

Figure 3.1: Diagrammatic representation of examples used to explain OWL-DL related concepts in this chapter

Throughout this chapter, refer to Figure 3.1 for an illustration of the main examples used. Here, classes are represented by rectangles, their names in bold. Data properties appear in regular-weight text inside the rectangles. Object properties are represented by dashed-line arrows. Inheritance is represented by solid-line arrows. Individuals are represented by rounded rectangles inside the *ABox*.

3.2.1 Concept Classes

OWL-DL classes are models of real-world concepts. The modeled concepts may be concrete (for instance, the concept of Person), or reified abstract concepts - concepts that are not tangible, such as a Loan or a Trip.

We use the terms *class* and *concept* interchangeably.

The way a class is described determines what types of individuals can be members of that class.

There are two ways of describing classes: as *atomic* (or *primitive*), or as *defined*. Atomic classes are declared, and membership in the class is explicitly defined. For instance, in our example we explicitly declare the concept AdministrativeStaff. Defined classes need necessary and sufficient conditions for membership. These conditions can be defined by properties, relationships (such as inheritance or disjointness), or operations (such as $\sqcap$ and $\sqcup$) on other classes. For instance, in our example we define the concept Manager as follows:

$$\text{Manager} \equiv \text{AdministrativeStaff} \sqcap \geq 1 \text{ manages}$$

That is, a Manager is any member of AdministrativeStaff who manages at least one project. (The difference between AdministrativeStaff and Manager is a subtle one: AdministrativeStaff contains individuals who *can* manage Projects, whereas Manager contains individuals who *do* manage Projects).

Multiple inheritance is possible in ontologies. For instance, we could define a concept TechnicalStaff like so:

$$\text{TechnicalStaff} \sqsubseteq \text{Employee} \sqcap \text{TechnicalExpert} \sqcap \neg \text{AdministrativeStaff}$$

This declares that all technical staff are both employees and technical experts.

We use the standard naming convention for concepts, which is to capitalize the first letter of every word and use no spaces, like so: *AdministrativeStaff*. Note that concepts, properties, and individuals should all use clear, descriptive names so that they are easily understood by users.

3.2.2 Properties

OWL Properties model real-world binary relationships between concepts and other concepts, or between concepts and values. Their equivalents in other conceptual models are slots (in frames [Minsky, 1981]), roles (in general DLs [Baader et al., 2007]), or relations (in UML [Fowler, 2003]).

There are two types of properties: *object* properties and *datatype* properties.

Object properties represent relationships between two classes; datatype properties represent relationships between classes and values. The relationship manages in the declaration AdminStaff manages Project is an object property. The relationship hasName in the declaration Employee hasName xsd:string is a datatype property.

Properties can have a number of characteristics, all typical of relations. These characteristics apply to individuals with these properties. The characteristics and examples of them are shown below:

- Functional: A functional property can only have a single value. For example, suppose we define a functional property hasNationality with domain Employee and range Nationality.

Then an instance of Employee can be related to at most one instance of Nationality via the hasNationality property; in other words, an employee can only have one nationality.

- Inverse functional: An inverse functional property can only be a single value. For example, suppose we define an inverse functional property isNationalityOf with domain Nationality and range Employee. isNationalityOf is the inverse property of hasNationality; since hasNationality is functional, isNationalityOf must be inverse functional. In this case, if two instances of isNationalityOf are related to the same instance of an Employee, then the two instances must be equal. In other words, once again, an employee can have only one nationality.

- Transitive: A transitive property, if applied to a sequence of classes, applies to any two classes in order. For example, suppose we define a transitive property reportsTo, with domain and range Employee. Then if employee A reports to employee B, and employee B reports to employee C, then employee A reports to employee C. Note that a transitive property cannot be functional.

- Symmetric: Symmetric properties apply to both their domain and range. For example, suppose we define a symmetric property worksWith with domain and range Employee. Then if employee A works with employee B, then employee B works with employee A.

- Antisymmetric: An antisymmetric property, as the name suggests, is not symmetric. For example, suppose we define an antisymmetric property reportsTo, with domain and range Employee. Then if employee A reports to employee B, then employee B cannot report to employee A.

- Reflexive: Reflexive properties can be applied to their domains. For example, suppose we define worksWith to be a reflexive property. Then we can say that employee A works with employee A.

- Irreflexive: Irreflexive properties cannot be applied to their domains. For example, suppose we define reportsTo to be irreflexive. Then we cannot say that employee A reports to employee A.

OWL properties can have a domain and a range defined. It is important to note that the domain and range are *not* treated the same as integrity constraints by most reasoners. If a property has domain X and we apply the property to individuals in the concept Y, most reasoners will infer that $Y \sqsubseteq X$. This is discussed further in Chapter 6.

We use the standard naming convention for properties, which is to use an asymmetric name (usually prefixed by *is* or *has*), use no spaces, and capitalize the first letter of every word except the first word, like so: *hasProjectStartDate*.

3.2.3 Individuals

Individuals are instances of concepts and model individual objects in the universe of discourse.

As with concepts, individuals can model concrete objects (e.g. *Anne*, an instance of Employee) or abstract concepts (e.g. *Canadian*, an instance of Nationality).

All individuals have a unique identifier and are countable. This fact is a key consideration when deciding whether to model something as an individual or a concept. For example, we choose to model Employee as a concept because it is likely we will have several instances of Employee with different characteristics (different names, email addresses, and relationships). *Anne* is one such instance. On the other hand, it is unlikely we would need several instances of a concept describing the nationality "Canadian". Thus we model it as an instance of the concept Nationality.

We use the standard naming convention for individuals, which is to capitalize the first letter of every word and use no spaces, like so: *JohnSmith*.

3.3 Reasoning on Ontologies

Reasoning on ontologies is the automatic classification of individuals, properties, and classes in the ontology. If we take the set of statements that define the classes, properties, and individuals in the ontology to be a set of *facts* about the universe of discourse, then reasoning uses these facts to infer *new* facts about the universe. In addition, given a specific statement about the universe of discourse (written in terms of the ontology), the reasoner can determine whether the statement is entailed by the ontology, that is, whether it is *consistent* with the terminology of and facts in the ontology.

The basic standard algorithm for reasoning on FOL languages is called *resolution*, first introduced in by Robinson [Robinson, 1965]. Resolution takes as input a knowledge base in conjunctive normal form[3] (CNF) and a query, also in CNF. It then uses a *refutation procedure* to determine whether the query is entailed by the knowledge base. Refutation procedures work by negating the query and searching for a contradiction. In resolution, clauses are successively resolved until an empty clause (the contradiction) is found. Resolution is explained in more detail in Chapter 4 of *Knowledge Representation and Reasoning* [Brachman and Levesque, 2004].

An example of resolution, also from Chapter 4 of [Brachman and Levesque, 2004] is shown in Figure 3.2. Suppose the knowledge base (KB) in Figure 3.2 describes some knowledge about a person. Thus what is known is that the person is a toddler; if the person is a toddler then the person is a child; if the person is a child and male then the person is a boy; if the person is an infant then the person is a child; if the person is a child and female, then the person is a girl; and the person is a female.

The example shows the steps taken in the resolution procedure to determine whether the person described by KB is a girl; that is, whether KB $\models$ Girl. For clarity, formulas are written in clausal

[3] In conjunctive normal form, clauses of disjunctions are joined by conjunctions. E.g. $(p \vee q) \wedge (x \vee y \vee z)$, which can also be written in clausal form, $\{[p, q], [x, y, z]\}$

CNF form (so that [¬Toddler, Child] is equivalent to saying Toddler ∨ Child). Clauses above the dotted line are taken from KB or the negated query [¬Girl]; clauses below the dotted line are those resolved (inferred) by the resolution procedure. Each resolved clause has two solid lines pointing to its inputs.

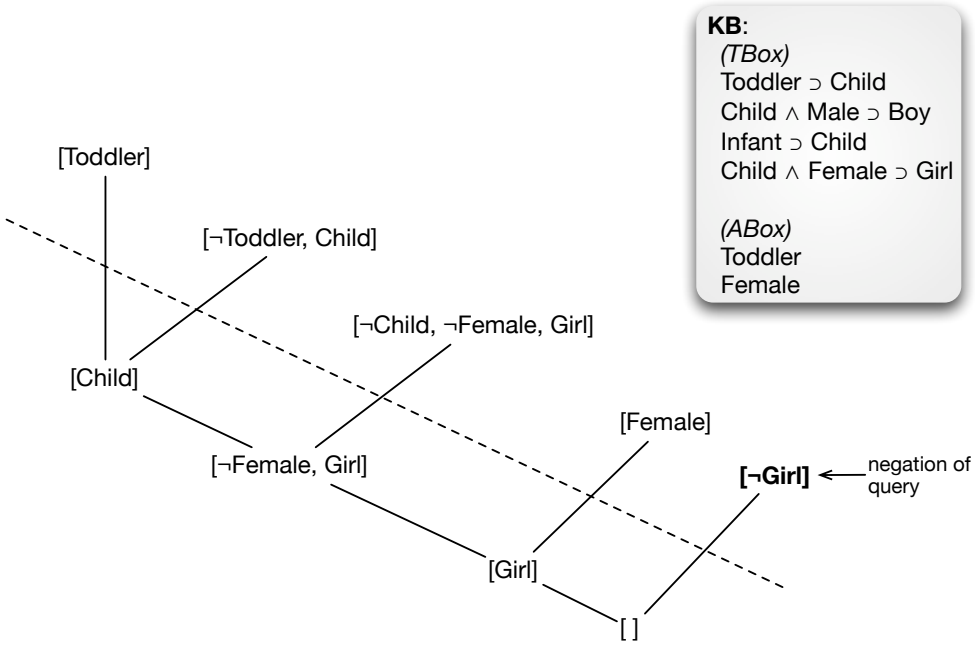

Figure 3.2: Example of resolution derivation, from Chapter 4 of [Brachman and Levesque, 2004]

The goal of resolution is to resolve clauses until the empty clause, [ ], is found. In the example, [ ] is not in KB or the query, so we find two clauses in KB that resolve to a new clause (one not already in KB). [Toddler] and [¬Toddler, Child] are two such clauses; they resolve to [Child]. This is because the formula Toddler ∧ (¬Toddler ∨ Child) implies Child. Next, by similar reasoning, we find that [Child] resolves with [¬Child, ¬Female, Girl] to produce the new clause, [¬Female, Girl]. This new clause resolves with [Female] to produce [Girl], which resolves with the negated query, [¬Girl] to produce the empty clause. Thus [¬Girl] produces a contradiction within KB, so KB entails [Girl], that is, KB $\models$ [Girl].

Note that this is a very simple example of resolution on FOL. In practise, resolution is more complicated, and resolution on full FOL is intractable. The resolution procedure has been optimized in various ways: for instance, a set of *inference rules* can be set to determine how and in what order clauses should be resolved. In addition, we can restrict the types of clauses in the knowledge base. When we do this we are in fact using a different representation language for the knowledge base, such as OWL-DL.

OWL-DL is not as expressive as full FOL, but reasoning on it is much easier. Reasoners for OWL-DL generally use a form of *tableau reasoning*, which was introduced shortly after resolution

by Smullyan and is explained in *First-order Logic* [Smullyan, 1995]. Tableau reasoning is a refutation procedure very similar to resolution. In tableaux reasoning, clauses are written in disjunctive normal form. The clauses are expanded into a tree, where each node of the tree represents a formula, and each branch represents the conjunction of its formulas. The tree is seen as a disjunction of branches. Tableaux reasoning, like resolution, uses a number of inference rules to find contradictions in each branch of the tree.

Reasoners can be optimized to work faster for the more common types of queries, often by changing the inference rules or the order the rules are applied in. Some well-known OWL-DL reasoners are Pellet [Parsia and Sirin, 2004], RACER [Haarslev and Möller, 2003], and FaCT++ [Tsarkov and Horrocks, 2006].

3.4 Conclusion

In this chapter we described the important concepts related to expressing ontologies in OWL-DL, and explained what it means to reason on an ontology. In the next chapters, we use these concepts to lay out a framework for schema mapping verification.

Chapter 4

The Mapping Verification Problem

This chapter explains what the mapping verification problem is and provides a formal definition of the problem we address in this thesis.

4.1 An Overview of the Mapping Verification Problem

Mapping verification occurs in the third and last step in the process of finding schema mappings between a source relational schema S and a target relational schema T. Recall from Figure 1.4 that the first step produces a multimapping $M : S \rightarrow T$ that matches pairs of elements in S and T based on lexical, structural, semantic, and statistical properties. The second step turns M into a set $M_{exec} : S \rightarrow T = \{\mu_1, ..., \mu_m\}$, where each $\mu_i \in M_{exec}$ is an executable mapping from S to T. Mappings in M_{exec} are ensured to be consistent with the constraints of the schemas (such as datatype definitions, primary and foreign key constraints, and null value restrictions).

The third step is the *mapping verification step*, in which the mappings in M_{exec} are checked for *semantic consistency*. We define semantic consistency in Section 4.3.2.

We focus on mappings between *relational schemas*. A relational schema is a set of *relations* $r(\mathbf{a})$, where $\mathbf{a}$ is an ordered list of attributes[1] $a_1, ..., a_n$. We say that a_i is *the attribute of* r *at position* i. Each attribute a_i can be associated with a real-world domain d_i.

The input to the mapping verification step consists of relational schemas S and T, as well as the results of the mapping creation step, M_{exec}. M_{exec} is a set of *executable schema mappings*, as defined below:

DEFINITION 1 (EXECUTABLE SCHEMA MAPPING) *Let S and T be relational schemas. An* executable schema mapping $\mu_{exec} : S \rightarrow T$ *is a set of* queries q *of the form:*

$$(\forall \mathbf{x})\mathbf{q_T}(\mathbf{x}) \leftarrow (\exists(\mathbf{y}))\mathbf{q_S}(\mathbf{x}, \mathbf{y})$$

where q_T is a relation in T and q_S is a conjunctive query over relations in S. Note that all variables that appear in q_T must appear in q_S.

[1] Note that relations correspond to *tables* and attributes correspond to *columns* in relational databases.

We restrict q_T to describe exactly one relation for simplicity. In addition, we will use Datalog-style notation to represent queries, so that quantifiers are implicit. This notation is explained in Chapter 5.

Our goal in the mapping verification step is to *semantically verify* each executable schema mapping $\mu \in M_{exec}$. Informally, semantically verifying μ involves determining whether the meanings of the elements in S are compatible with the meanings of the elements in T to which they are mapped byμ.

4.1.1 Example

To illustrate the mapping verification problem, we revisit an example we introduced in Chapter 1, as shown in Figure 4.1.

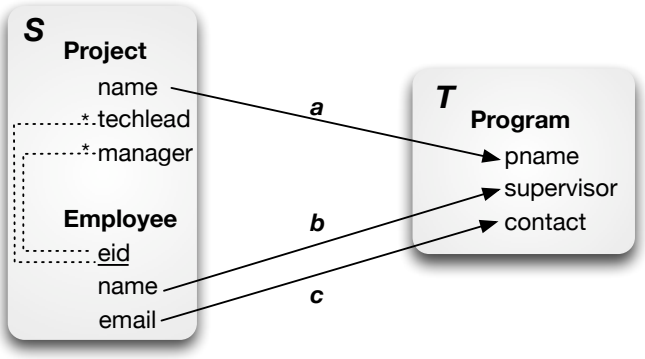

Figure 4.1: An example data exchange scenario, modified from the running example in [Alexe et al., 2008]. A schema mapping is to be found from schema S to the schema T. Arrows denote correspondences between the schemas found in the schema matching step.

The figure shows the source schema S and the target schema T. The arrows denote the results of the matching step of the schema mapping process, the multimapping $M : S \rightarrow T$. M consists of the following pairs:

a: (S.`Project.name`, T.`Program.pname`)
b: (S.`Employee.name`, T.`Program.supervisor`)
c: (S.`Employee.email`, T.`Program.contact`)

The correspondences in M are used to generate possible schema mappings. Clio [Popa et al., 2002] creates schema mappings by using *logical relations* to "understand" the correspondences. Logical relations are groups of one or more relations in a schema that are joined using primary and foreign key constraints defined in the schema. These logical relations suggest to Clio which correspondences should be interpreted together.

Based upon the logical relations present in S and T, the list of mappings $M_{exec} : S \rightarrow T$ that a schema mapping generation tool like Clio might find are shown below. Note that, for simplicity,

we chose an example in which the possible mappings consist of only one query each; in real applications mappings would consist of more than one query. Also note that we use the Datalog-style notation here rather than Clio's notation.

μ_1 : $q_1 : T.\texttt{Program}(x_1, x_2, x_3) \leftarrow S.\texttt{Project}(x_1, y, -), S.\texttt{Employee}(y, x_2, x_3)$
(Denotes that the supervisor and contact fields in T are filled by the technical leader's name and email in S)

μ_2 : $q_2 : T.\texttt{Program}(x_1, x_2, x_3) \leftarrow S.\texttt{Project}(x_1, -, y), S.\texttt{Employee}(y, x_2, x_3)$
(Denotes that the supervisor and contact fields in T are filled by the manager's name and email in S)

μ_3 : $q_3 : T.\texttt{Program}(x_1, x_2, x_3) \leftarrow S.\texttt{Project}(x_1, y, z), S.\texttt{Employee}(y, x_2, x_3), S.\texttt{Employee}(z, x_2, x_3)$
(Denotes that the supervisor and contact fields in T are filled by the technical leader's and manager's name and email in S; two entries in `Program` are created for each project.)

In order to *semantically verify* these queries, we must determine whether T`.Program.supervisor` refers to the name of the technical leader of a project (S`.Project.techlead`), or of the manager (S`.Project.manager`). We must also determine whether T`.Program.contact` refers to the email address of the technical leader of a project or of the manager.

In this example, in the source schema S, `techlead`s and `manager`s are disjoint. The designers of the target schema T intended for `supervisor` to refer to the administrative leader of a program (the manager), not the technical leader.

Notice that there is no way to determine this information - the *intent* of the target schema designers - without using information about the schema elements beyond the schema itself, such as schema documentation, a knowledge base, clues in the data instances, or the schema designer's knowledge about the schema (which requires user intervention).

4.2 Representing Semantics

Before we can formally define the mapping verification problem, it is important to determine how to find the meaning of schema elements. In other words, we need to find a way to harness the intent and knowledge the schema designers had when designing the schemas.

We represent knowledge about the domain using a domain ontology. Ontologies were described in Chapter 3. We now formally define an ontology as follows:

DEFINITION 2 (DOMAIN ONTOLOGY) *A* domain ontology $\mathcal{O}$ *is a description of a domain* d*.* $\mathcal{O}$ *consists of a set* $\{\mathcal{C}, \mathcal{P}, \mathcal{I}\}$ *of concepts* $\mathcal{C}$*, properties* $\mathcal{P}$ *and individuals* $\mathcal{I}$*. Every ontology has a concept* Thing $\in \mathcal{C}$*, which we denote by* $\top$*, that subsumes all other concepts.*

The domain ontology used for mapping verification should describe the same domain as the schemas being mapped.

In order to describe the meanings of attributes in a schema using a domain ontology, we use *semantic annotations*.

DEFINITION 3 (SEMANTIC ANNOTATION MAPPING) *Let* $r(\mathbf{a})$, $\mathbf{a} = a_1, ..., a_n$ *be a relation in a schema* S *and* $\mathcal{O} = \{\mathcal{C}, \mathcal{P}, \mathcal{I}\}$ *be an ontology that partially covers the domain of* S. *A* semantic annotation mapping *for* r *is a partial function*

$$\alpha_r : \{1, ..., n\} \rightarrow \mathcal{C}$$

Note that a semantic annotation mapping is a *partial* mapping, which means that we do not require that *every* attribute in a schema be annotated. Thus we do not require $\mathcal{O}$ to contain at least one concept or property that corresponds to *every* attribute in the schema; however, the more corresponding concepts and properties that $\mathcal{O}$ contains, the more meaningful the semantic annotation mapping can be.

DEFINITION 4 (SEMANTIC ANNOTATION) *Let* $S = \{r_1, ..., r_k\}$ *be a relational schema with k relations, such that each* r_i *has a semantic annotation mapping* α_{r_i}. *We call a set*

$$\alpha^S = \{\alpha_{r_1}, ..., \alpha_{r_k}\}$$

a semantic annotation *for* S.

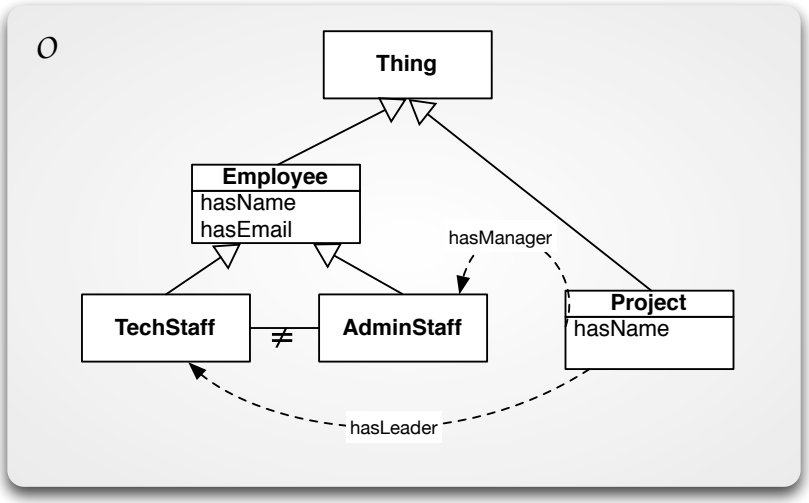

Figure 4.2: A sample ontology

Suppose we have an ontology $\mathcal{O}$, like the one shown in Figure 4.2. A semantic annotation α^S for the schema S described in Figure 4.1 might look like the following:

$\alpha_{\texttt{Project}} : \{2 \rightarrow \mathsf{TechStaff}, 3 \rightarrow \mathsf{AdminStaff}\}$
$\alpha_{\texttt{Employee}} : \{1 \rightarrow \mathsf{Employee}, 2 \rightarrow \mathsf{Employee}, 3 \rightarrow \mathsf{Employee}\}$

A semantic annotation α^T for the schema T described in Figure 4.1 might look like the following:

$$\alpha_{\texttt{Program}} : \{2 \rightarrow \mathsf{AdminStaff}, 3 \rightarrow \mathsf{AdminStaff}\}$$

Notice how much background information is contained in these semantic annotations. The ontology $\mathcal{O}$ in Figure 4.2 describes the domain of S and T. It states that technical staff (TechStaff) and administrative staff (AdminStaff) are both a type of Employee, and that TechStaff and AdminStaff are disjoint concepts.

The semantic annotation for S (α^S) tells us that the `Employee` relation corresponds exactly to the Employee concept in $\mathcal{O}$; in the `Project` relation, `techlead` corresponds to the concept TechStaff and `manager` corresponds to the concept AdminStaff; and from the information contained in $\mathcal{O}$ we know that a `techlead` cannot be a `manager` and vice versa because TechLead and AdminStaff are disjoint.

The semantic annotation for T (α^T) clarifies that `supervisor` and `contact` are associated with the AdminStaff concept, which means they do not refer to the TechStaff concept. Thus α^T captures the *intent* behind T's schema designers' attribute labeling choices.

Notice also the flexibility of semantic annotations: although our ontology has a concept Project that corresponds to the S`.Project` and T`.Program` relations, it is not *necessary* to express this fact in the annotations.

4.3 Semantic Consistency of Schema Mappings

We use the notion of *semantic consistency* of a schema mapping to semantically verify a mapping. For a mapping to be semantically consistent, the concepts associated with variables on the left hand side of the mapping must match those associated with the same variables on the right hand side. The definitions below explain this concept more precisely.

4.3.1 Associated Concepts

DEFINITION 5 ($\gamma_r(v, j)$) *Let r (**a**), $\mathbf{a} = a_1, ..., a_n$ be a subgoal in a conjunctive query in which the variable v appears. The* concept associated with v in r at position j *is*

$$\gamma_r(v, j) = \begin{cases} \alpha_r(j) & \text{if } \alpha_r(j) \text{ is defined} \\ \top & \text{otherwise} \end{cases}$$

DEFINITION 6 ($\gamma_r(v)$) *Now let $j_1, ..., j_k$ be all positions in r where v appears. The* concept asso-

ciated with v in r *is:*

$$\gamma_r(v) = \begin{cases} \gamma_{\,r}(v, j_1) \sqcap ... \sqcap \gamma_r(v, j_k) & \textit{if } v \textit{ appears in } r \\ \top & \textit{otherwise} \end{cases}$$

Accounting for Key Constraints

In the example in Section 4.1.1, S.`Employee.eid` is a primary key for S.`Employee.name` and S.`Employee.email`. Suppose we have the semantic annotation α^S defined at the end of Section 4.2, and suppose we have a conjunctive query:

$$q : S.\texttt{Project}(x_1, y, -), S.\texttt{Employee}(y, x_2, x_3)$$

Consider the variable x_2, which refers to the attribute S.`Employee.name`. $\alpha_{\texttt{Employee}}$ associates this attribute with the concept Employee. However, we can see that x_2 actually refers to the name of a technical leader. This is because the primary key for x_2 in `Employee` is `Employee.eid`, represented by y, and y also represents `Project.techlead` in q_1. `Project.techlead` is associated with the concept TechStaff. Thus x_2 should be associated with the concept Employee $\sqcap$ TechStaff.

We take key constraints into account when determining the concept associated with a variable in a query.

DEFINITION 7 ($\delta_r(j)$) *Let r ($\mathbf{a}$), $\mathbf{a} = a_1, ..., a_n$ be a relation, and $a_{i_1}, ..., a_{i_k} \rightarrow a_j$ be a primary key constraint over $\mathbf{a}$. The* key constraint set of position j in r, $\delta_r(j)$, *is the set* $\{i_1, ..., i_k\}$

DEFINITION 8 ($\Delta_r(v)$) *Let q_X be a conjunctive query over a schema X and let r ($\mathbf{v}$), $\mathbf{v} = v_1, ..., v_n$ be a subgoal in q_X. Let v be a variable in q_X that appears in position i in r ($\mathbf{v}$). The* key constraint set of v in r, $\Delta_r(v)$, *is the set of variables* $\{u_1, ..., u_k\}$ *such that:*

- u_j *is a variable in q_X at position j in r ($\mathbf{v}$).*
- $j \in \delta_r(i)$

For example, for the query q, $\Delta_{\texttt{Employee}}(x_2) = \{y\}$ because y refers to `Employee.eid`, the primary key of `Employee`.

Associated Concepts in Queries

We extend our previous definitions of concepts associated with variables in relations to conjunctive queries, taking into account key constraints.

DEFINITION 9 ($\Gamma_q(v)$) *Let $q = r_1(\mathbf{a}_1), ..., r_k(\mathbf{a}_k)$ be a conjunctive query in which the variable v appears. The* concept associated with v in q , $\Gamma_q(v)$ *is defined as:*

$$\Gamma_q(v) = \gamma'_{r_1}(v) \sqcap ... \sqcap \gamma'_{r_k}(v)$$

where

$$\gamma'_{r_i}(v) = \gamma_{r_i}(v) \sqcap \bigsqcap_{u \in \Delta_{r_i}(v)} \Gamma_q(u)$$

For example, using the query q and semantic annotation α^S, we can define the following associated concepts for the variables y and x_2:

$\Gamma_q(y) = \gamma_{\texttt{Project}}(y) \sqcap \gamma_{\texttt{Employee}}(y) = \mathsf{TechStaff} \sqcap \mathsf{Employee}$
(Note that $\Delta_{\texttt{Project}}(y) = \Delta_{\texttt{Employee}}(y) = \emptyset$, so its associated concept in q is simply a conjunction of its associated concepts in the two relations in which it appears.)

$\gamma_{\texttt{Employee}}(x_2) = \gamma_{\texttt{Employee}}(x_2, 2) = \mathsf{Employee}$
(This is because the semantic annotation mapping $\alpha_{\texttt{Employee}}$ states that the attribute in position 2 of `Employee`, `Employee.name`, is associated with the concept Employee.)

$\Gamma_q(x_2) = \gamma_{\texttt{Employee}}(x_2) \sqcap \Gamma_q(y) = \mathsf{Employee} \sqcap (\mathsf{TechStaff} \sqcap \mathsf{Employee})$
(Note that $\Delta_{\texttt{Employee}}(x_2) = y$ because `Employee.eid` is the primary key for `Employee`).

4.3.2 Semantic Consistency

DEFINITION 10 (SEMANTIC CONSISTENCY OF VARIABLES) *Let $q : q_T \leftarrow q_S$ be a query in a mapping from a source schema S to a target schema T, and let α^S and α^T be their semantic annotations with respect to an ontology $\mathcal{O}$. If the concept associated with a variable v in q_S is $\Gamma_{q_S}(v)$ and the concept associated with v in q_T is $\Gamma_{q_T}(v)$ then q states that:*

$$\mathcal{O} \models \Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v) \tag{4.1}$$

We say that v in q is consistent with the semantics of S and T with respect to $\mathcal{O}$ *if and only if statement 4.1 is true.*

DEFINITION 11 (SEMANTIC CONSISTENCY OF QUERIES) *We say that q* is consistent with the semantics of S and T with respect to $\mathcal{O}$ *if $\forall v \in q$, v in q is consistent with the semantics of S and T with respect to $\mathcal{O}$.*

DEFINITION 12 (SEMANTIC CONSISTENCY OF SCHEMA MAPPINGS) *Let $\mu : S \rightarrow T$ be a schema mapping and $\mathcal{O}$ be a domain ontology. We say that μ* is consistent with the semantics of S and T with respect to $\mathcal{O}$ *if $\forall q \in \mu$, q is consistent with the semantics of S and T with respect to $\mathcal{O}$.*

4.4 A Formal Definition of the Mapping Verification Problem

Given the above definitions, we can now formally define the mapping verification problem when knowledge is represented using a domain ontology. Suppose we have

- Two relational schemas S and T;
- An ontology $\mathcal{O}$ that partially covers the domain of $S \cup T$;
- A semantic annotation of S in terms of $\mathcal{O}$, α^S;
- A semantic annotation of T in terms of $\mathcal{O}$, α^T; and
- An executable mapping from S to T, $\mu : S \rightarrow T$.

The *mapping verification problem for* μ is to determine whether μ is consistent with the semantics of T.

4.5 Conclusion

In this chapter we formally defined the concepts related to mapping verification, then used these to formally define the mapping verification problem. In the next chapter, we present a framework and its algorithms for solving the mapping verification problem.

Chapter 5

A Framework for Mapping Verification

We now present our framework for mapping verification and describe two algorithms: our main algorithm for verifying the semantic consistency of schema mappings, and one for finding concepts associated with variables in a query based on semantic annotation mappings. We also discuss a method for obtaining semantic annotations semi-automatically.

5.1 Our Framework

Figure 5.1 shows an overview of our mapping verification framework.

5.1.1 Input

As input, the framework takes two relational schemas[1] S and T, a domain ontology $\mathcal{O}$, a reasoner ***R*** compatible with $\mathcal{O}$, a set of semantic annotation mappings from S to $\mathcal{O}$, α^S, a set of semantic annotation mappings from T to $\mathcal{O}$, α^T, as well as a single executable mapping $\mu_{exec} : S \rightarrow T$ (a set of conjunctive queries of the form $q : q_T \leftarrow q_S$).

Given a list of executable mappings $M_{exec} : S \rightarrow T$ (the output from the second step of the process of finding schema mappings - see Figure 1.4 for a general overview of this process) our framework is applied to each executable mapping in turn.

5.1.2 Checking Semantic Consistency of a Schema Mapping

Our framework checks the semantic consistency of the mapping by checking the semantic consistency of each query in the mapping with respect to $\mathcal{O}$. This process is outlined in our main mapping verification algorithm, Algorithm 1. We break each query q into two parts: the head q_T and the body q_S. We then use an algorithm (Algorithm 3) to find Γ_{q_S} and Γ_{q_T}, the concepts associated with the variables in q_T and q_S based on the semantic annotation mappings we were given.

[1] We use only the schemas, not the data present in the databases.

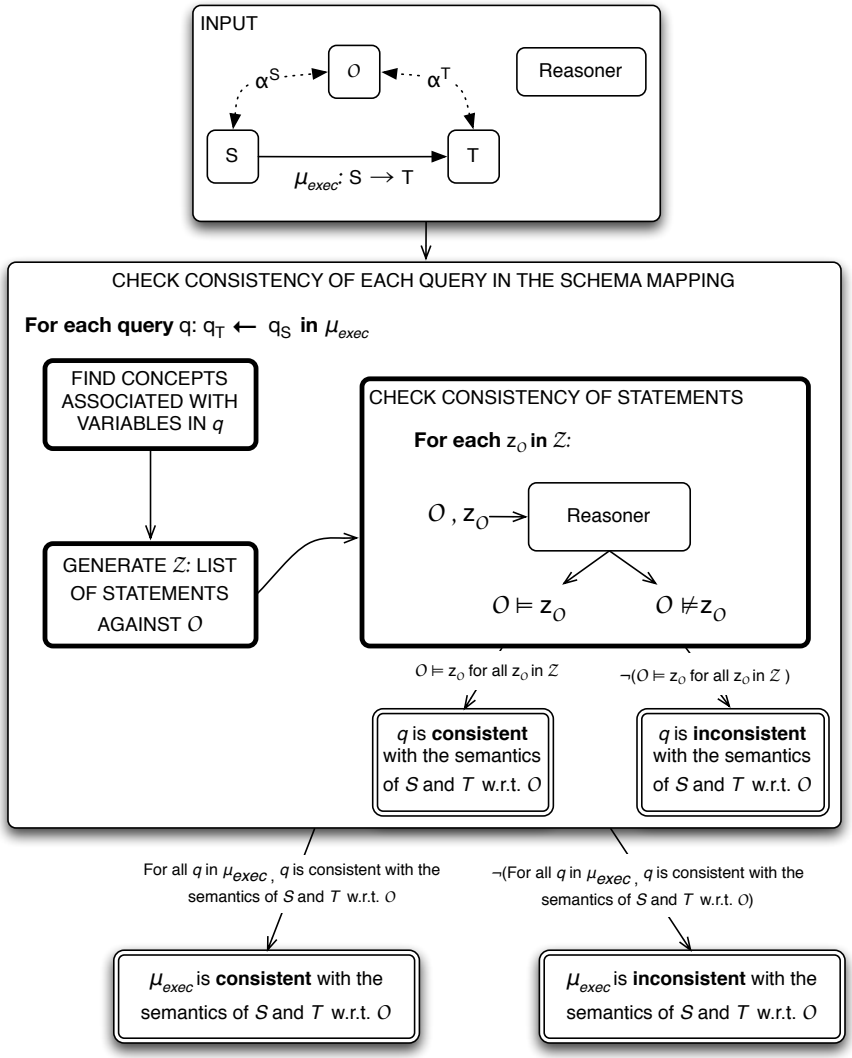

Figure 5.1: Our mapping verification framework

Using Γ_{q_S} and Γ_{q_T}, we rewrite q as a list of statements written in terms of the ontology (see Algorithm 2). In essence, if a_1 in S is associated with a concept C_1 in $\mathcal{O}$, a_2 in T is associated with a concept C_2 in $\mathcal{O}$, and q maps a_1 into a_2, then q implies that the statement $C_1 \sqsubseteq C_2$ is satisfied in $\mathcal{O}$. Intuitively, data under a_1 is a "kind" of C_1, and data under a_2 is a "kind" of C_2; because q moves data from a_1 to a_2, it implies that a_2 accepts data of kind C_1, so C_1 must be a kind of C_2.

Our framework uses a reasoner to verify the validity of these statements. If all the statements are satisfied in $\mathcal{O}$, then we say that q is consistent with the semantics of S and T with respect to $\mathcal{O}$. Otherwise, we say it is inconsistent.

Similarly, μ_{exec} is consistent with the semantics of S and T with respect to $\mathcal{O}$ if and only if each query q in μ_{exec} is consistent with the semantics of S and T with respect to $\mathcal{O}$.

The schema mapping designer uses this information to choose a final, executable mapping

$\mu_{exec} : S \rightarrow T$, which can be used to transfer the data from the source database S to the target database T.

5.1.3 Representation and Notation

S and T are relational schemas and we assume they will be input expressed in standard SQL notation. However, when we refer to relations inside our framework (for example in queries or semantic annotation mappings), we represent them in the form r ($\mathbf{a}$), that is, the relation name followed by an ordered list of attributes, for example: `Project(pname, techlead, manager)`.

The ontology $\mathcal{O}$ was formally defined by Definition 2 in Chapter 4. We assume $\mathcal{O}$ is defined using OWL-DL, which was explained in Chapter 3.

The semantic annotation mappings between the schemas and the ontology are written as a set of partial mappings, one for each relation in the schema. Positions of attributes in the relation are mapped to concepts in the ontology. Note that, because this is a partial mapping, not every position needs to be mapped to a concept. A sample semantic annotation mapping, from our running example, can be expressed as follows:

$$\alpha_{\texttt{Project}} : \{2 \rightarrow \textsf{TechStaff}, 3 \rightarrow \textsf{AdminStaff}\}$$

$\alpha_{\texttt{Project}}$ maps the attribute at position 2 of `Project`, `techlead`, to the concept TechStaff in the ontology. This suggests that the meaning of `techlead` is most closely related to the meaning of the concept TechStaff. $\alpha_{\texttt{Project}}$ maps the attribute at position 3, `manager`, to the concept AdminStaff. This suggests that the meaning of `manager` is most closely associated with the meaning of the concept AdminStaff. The attribute at position 1, `name`, is not mapped at all. This suggests that its meaning is unknown or deemed unnecessary to specify.

The mapping μ is a set of queries, each of which is also expressed as a Datalog-style query, $q : q_T \leftarrow q_S$, where q_T is a conjunctive query over T and q_S is a conjunctive query over S. q_T contains a single relation r in T, where each attribute in r is represented by a unique variable name. If a variable v represents an attribute in the relation r in q_T, we say that v *occurs in* q_T.

q_S contains a conjunction of relations in S, where each attribute in each relation in q_S is represented by a variable - either a variable that occurs in q_T, or some other unique variable, often representing a join or data that is not moved. Note that q must meet the safety condition for Datalog - that is, each variable that occurs in q_T must occur in q_S.

An example query that could be made over our running example is one that maps technical leaders (`techlead`) in S to supervisors and contacts in T, as follows:

q: $T.\texttt{Program}(x_1, x_2, x_3) \leftarrow S.\texttt{Project}(x_1, y, z), S.\texttt{Employee}(y, x_2, x_3)$

5.1.4 Example

To illustrate how our framework runs, let us introduce an expanded version of our running example. Figure 5.2 shows our example schemas semantically annotated using a domain ontology. The semantic annotations of S and T are represented by dotted lines.

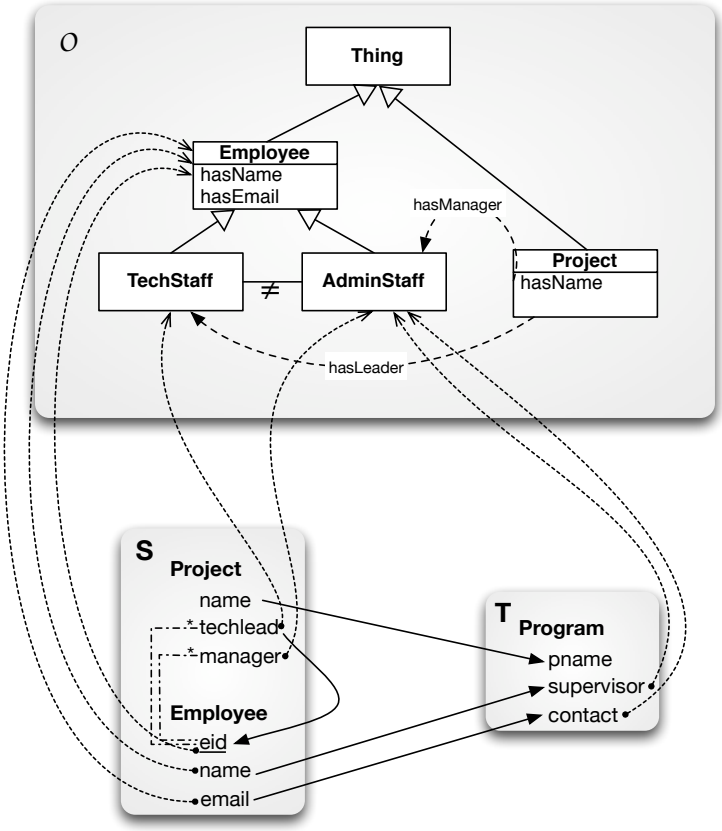

Figure 5.2: Our example schema mapping scenario, now annotated with an ontology

Let us describe the input to our framework from this example. We begin with the source schema S, with two relations `Project` and `Employee`; and the target schema T, with a single relation `Program`. These relations are represented in standard SQL notation, but when used in queries or semantic annotation mappings, we write them in the form:

S.`Project(name, techlead, manager)`
S.`Employee(eid, name, email)`
T.`Program(pname, supervisor, contact)`

Note that the order of attributes is important, so the attribute `techlead` is known as the attribute in position 2 of the relation `Project`.

The next input is an ontology $\mathcal{O}$. $\mathcal{O}$ contains a concept Employee with two data properties,

hasName and hasEmail, and two disjoint children TechStaff and AdminStaff. $\mathcal{O}$ also contains the universal concept Thing (denoted $\top$), which subsumes all other concepts in $\mathcal{O}$. For the sake of clarity we use a UML-style notation to display the ontology in Figure 5.2. Concept names are at the top of the boxes, data properties are below concept names, and object properties are the relationships between concepts.

Next we input a semantic annotation α^S for S and a semantic annotation α^T for T. Notice once again that semantic annotation mappings are partial, so not all attributes are annotated.

α^S consists of two semantic annotation mappings: $\alpha_{\texttt{Project}}$ and $\alpha_{\texttt{Employee}}$. These are represented as follows:

$$\alpha_{\texttt{Project}} : \{2 \rightarrow \mathsf{TechStaff}, 3 \rightarrow \mathsf{AdminStaff}\}$$
$$\alpha_{\texttt{Employee}} : \{1 \rightarrow \mathsf{Employee}, 2 \rightarrow \mathsf{Employee}, 3 \rightarrow \mathsf{Employee}\}$$

α^Tconsists of one semantic annotation mapping, $\alpha_{\texttt{Program}}$, which looks like the following:

$$\alpha_{\texttt{Program}} : \{2 \rightarrow \mathsf{AdminStaff}, 3 \rightarrow \mathsf{AdminStaff}\}$$

We are also given two executable mappings from S to T:

$$\mu_1 : q_1\text{:} \quad T.\texttt{Program}(x_1, x_2, x_3) \quad \leftarrow \quad S.\texttt{Project}(x_1, y, z),\, S.\texttt{Employee}(y, x_2, x_3)$$
$$\mu_2 : q\text{:} \quad T.\texttt{Program}(x_1, x_2, x_3) \quad \leftarrow \quad S.\texttt{Project}(x_1, y, z),\, S.\texttt{Employee}(z, x_2, x_3)$$

Let us show how our framework verifies μ_1, which is denoted by the solid arrows in Figure 5.2. μ_1 has only one query, q_1. This query maps the `name` and `email` of `techlead` in `Project` into the `supervisor` and `contact` attributes of `Program`. To verify μ_1, our framework will determine whether or not q_1 is consistent with the semantics of S and T with respect to $\mathcal{O}$.

The first step for our framework is to find the concepts associated with the variables in the head and body of q. This is done using Algorithm 3, and results in the following:

$$\Gamma_{qS}(x_1) = \top \qquad \Gamma_{qT}(x_1) = \top$$
$$\Gamma_{qS}(x_2) = \mathsf{Employee} \sqcap \mathsf{TechStaff} \qquad \Gamma_{qT}(x_2) = \mathsf{AdminStaff}$$
$$\Gamma_{qS}(x_3) = \mathsf{Employee} \sqcap \mathsf{TechStaff} \qquad \Gamma_{qT}(x_3) = \mathsf{AdminStaff}$$
$$\Gamma_{qS}(y) = \mathsf{Employee} \sqcap \mathsf{TechStaff}$$
$$\Gamma_{qS}(z) = \mathsf{AdminStaff}$$

The next step is to use Algorithm 2 to rewrite our query q_1 in terms of these associated concepts. One statement is created for each variable v in q_T, and each is of the form $\Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v)$. For our example, the statements generated are:

$$x_1 : \top \sqsubseteq \top$$
$$x_2 : \mathsf{Employee} \sqcap \mathsf{TechStaff} \sqsubseteq \mathsf{AdminStaff}$$
$$x_3 : \mathsf{Employee} \sqcap \mathsf{TechStaff} \sqsubseteq \mathsf{AdminStaff}$$

The first statement, $\top \sqsubseteq \top$ is discarded immediately by Algorithm 2 because it is trivial - it will always be satisfied by $\mathcal{O}$.

The remaining two statements are returned by Algorithm 2. At this point, the reasoner is called to check whether the statements for x_2 and x_3 are satisfied by $\mathcal{O}$. When the reasoner checks the statement $\mathsf{Employee} \sqcap \mathsf{TechStaff} \sqsubseteq \mathsf{AdminStaff}$, it finds that it is not satisfied by $\mathcal{O}$; this is because the statement is equivalent to the statement $\mathsf{TechStaff} \sqsubseteq \mathsf{AdminStaff}$, which is false because in $\mathcal{O}$, TechStaff and AdminStaff are specified to be disjoint.

Thus we see that q_1 is *not* consistent with the semantics of S and T with respect to $\mathcal{O}$. Essentially, q_1 is trying to map technical leaders in S to supervisors in T, but the semantic annotation for T clearly specifies that supervisors are administrative staff, not technical staff.

Based on this result, our framework concludes that μ_1 is not consistent with the semantics of S and T with respect to $\mathcal{O}$.

Now let us show how our framework verifies μ_2. μ_2 also has only one query, q_2. This query maps the `name` and `email` of `manager` in `Project` into the `supervisor` and `contact` attributes of `Program`. To verify μ_2, our framework will determine whether or not q_2 is consistent with the semantics of S and T with respect to $\mathcal{O}$.

Once again, the first step is to find the concepts associated with the variables in the head and body of q. This is done using Algorithm 3, and results in the following:

$$\begin{array}{ll} \Gamma_{qS}(x_1) = \top & \Gamma_{qT}(x_1) = \top \\ \Gamma_{qS}(x_2) = \mathsf{Employee} \sqcap \mathsf{AdminStaff} & \Gamma_{qT}(x_2) = \mathsf{AdminStaff} \\ \Gamma_{qS}(x_3) = \mathsf{Employee} \sqcap \mathsf{AdminStaff} & \Gamma_{qT}(x_3) = \mathsf{AdminStaff} \\ \Gamma_{qS}(y) = \mathsf{TechStaff} & \\ \Gamma_{qS}(z) = \mathsf{Employee} \sqcap \mathsf{AdminStaff} & \end{array}$$

The next step is to use Algorithm 2 to rewrite q_2 in terms of these associated concepts. One statement is created for each variable v in q_T, and each is of the form $\Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v)$. For our example, the statements generated are:

$$\begin{array}{l} x_1 : \top \sqsubseteq \top \\ x_2 : \mathsf{Employee} \sqcap \mathsf{AdminStaff} \sqsubseteq \mathsf{AdminStaff} \\ x_3 : \mathsf{Employee} \sqcap \mathsf{AdminStaff} \sqsubseteq \mathsf{AdminStaff} \end{array}$$

The first statement, $\top \sqsubseteq \top$ is discarded immediately by Algorithm 2 because it is trivial - it will always be satisfied by $\mathcal{O}$.

The remaining two statements are returned by Algorithm 2. At this point, the reasoner is called to check whether the statements for x_2 and x_3 are satisfied by $\mathcal{O}$. When the reasoner checks the statement for x_2, $\mathsf{Employee} \sqcap \mathsf{AdminStaff} \sqsubseteq \mathsf{AdminStaff}$, it finds that it is equivalent to the statement $\mathsf{AdminStaff} \sqsubseteq \mathsf{AdminStaff}$ because $\mathsf{AdminStaff} \sqsubseteq \mathsf{Employee}$. Thus it finds that the statement is satisfied by $\mathcal{O}$. The statement for x_3 is identical.

Since all of the generated statements are satisifed by $\mathcal{O}$, we say that q_2 *is* consistent with the semantics of S and T with respect to $\mathcal{O}$.

This makes sense because q_2 maps managers in S to supervisors in T, and the intended meaning of `supervisor` is to denote administrative staff such as managers.

Based on this result, our framework concludes that μ_2 is consistent with the semantics of S and T with respect to $\mathcal{O}$.

5.2 Algorithms

We now present the algorithms used in our framework. In the Section 5.2.1 we describe two algorithms used to check the semantic consistency of a single mapping from a source schema to a target schema; in Section 5.2.2 we describe how we find concepts associated with variables in a query.

5.2.1 Mapping Verification Algorithm

Algorithm 1: SemanticallyVerifySchemaMapping

input: Ontology $\mathcal{O}$, source schema S, target schema T, semantic annotation α^S, semantic annotation α^T, mapping $\mu : S \rightarrow T$, reasoner ***R***
output: **true** if μ is consistent with the semantics of S and T with respect to $\mathcal{O}$, **false** otherwise

local map Γ_{q_S}, map Γ_{q_T}, list $\mathcal{Z}$, boolean *isConsistent*
isConsistent := **true**

foreach *query* $q : q_T \leftarrow q_S$ *in* μ **do**
  $\mathcal{Z}, \Gamma_{q_S}, \Gamma_{q_T} := \emptyset$
  Let $\mathcal{V}_T$ be the set of variables occurring in q_T
  foreach *variable* $v \in \mathcal{V}_T$ **do**
    $\Gamma_{q_T}(v)$:= FindAssociatedConceptsInQuery (v, q_T, α^T)
  end
  Let $\mathcal{V}_S$ be the set of variables occurring in q_S
  foreach *variable* $v \in \mathcal{V}_S$ **do**
    $\Gamma_{q_S}(v)$:= FindAssociatedConceptsInQuery (v, q_S, α^S)
  end
  $\mathcal{Z}$:= GenerateVerificationStatements $(\Gamma_{q_S}, \Gamma_{q_T})$
  foreach *statement* z *in* $\mathcal{Z}$ **do**
    isConsistent := ***R***.CheckConsistency $(z, \mathcal{O}) \wedge$ *isConsistent*
  end
end
return *isConsistent*

Algorithm 1 is our main algorithm for semantically verifying a schema mapping from a source schema S to a target schema T using semantic annotations α^S and α^T in terms of an ontology $\mathcal{O}$ and a reasoner ***R***.

The goal of this algorithm is to *semantically verify* the mapping μ; in other words, to check whether μ is consistent with the semantics of S and T with respect to $\mathcal{O}$. Recall that μ is a set of

queries which map a conjunctive query of relations in S to individual relations in T. A mapping is semantically verified if all queries inside it are semantically verified; thus Algorithm 1 verifies each query individually.

To semantically verify a query $q : q_T \leftarrow q_S$, the algorithm checks that the concepts associated with the variables in q_T (the head of q) subsume the concepts associated with the variables in q_S (the body of q).

We first find Γ_{q_T}, a map that associates concepts in $\mathcal{O}$ with variables in q_T, and Γ_{q_S}, a map that associates concepts in $\mathcal{O}$ with variables in q_S. The procedure for finding these concepts makes use of the semantic annotations for S and T, and is outlined in Section 5.2.2. Recall that a conjunction of concepts is also a concept; thus, each variable may be mapped to a concept or a conjunction of concepts in $\mathcal{O}$. Since semantic annotation mappings are only partial mappings, some variables may not be associated with a specific concept; in this case we assume they are associated with $\top$, the universal concept that subsumes all other concepts.

Now that we have Γ_{q_T} and Γ_{q_S}, we essentially know the assigned *meaning* of each attribute a in S and T. The next step is to check that, if q maps an attribute a_S in S to an attribute a_T in T, that the meanings of a_S and a_T are compatible.

To do this, Algorithm 1 uses Algorithm 2 to check whether Γ_{q_T} and Γ_{q_S} associate the exact same concepts with the same variables and, in those cases where they do not, to generate a list of verification statements of the form $\Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v)$.

Algorithm 1 then inputs these statements to a reasoner, which checks whether they are satisfied by the ontology $\mathcal{O}$. (Reasoning is explained in Chapter 3.) If any of the statements is *not* satisfied, this indicates that the meanings of the attributes represented by v are incompatible, thus the query is not consistent with the semantics of S and T with respect to $\mathcal{O}$, and neither is the mapping μ.

The algorithm returns a boolean variable *isConsistent*. If *isConsistent* is true, then μ is consistent with the semantics of S and T with respect to $\mathcal{O}$; otherwise it is not.

Algorithm 2: GenerateVerificationStatements

input: map Γ_{q_S}, map Γ_{q_T}
output: list of statements $\mathcal{Z}$
local list $\mathcal{Z} := \emptyset$

foreach *variable* v *in* Γ_{q_T} *where* $\Gamma_{q_T}(v) \neq \top$ **do**
  if $\Gamma_{q_T}(v) \neq \Gamma_{q_S}(v)$ **then**
    $z := \Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v)$
    add z to $\mathcal{Z}$
  end
end
return $\mathcal{Z}$

Algorithm 2 uses the two maps Γ_{q_S} and Γ_{q_T} to generate a list of statements about the concepts that are implied by the query q. Essentially, we want to rewrite the original query as a list of statements in terms of the concepts associated with its variables. Since q maps variables in S into

variables in T and not vice versa, each statement is of the form $\Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v)$; that is, q implies that the attribute referred to by v in S has a meaning that is subsumed by the meaning of the attribute referred to by v in T.

The algorithm iterates over all variables in Γ_{q_T}. This is because all variables in Γ_{q_T} must occur in Γ_{q_S} (based on the safety condition of Datalog, as discussed earlier).

For each variable v in Γ_{q_T}, the goal of Algorithm 2 is to generate a statement that is not trivial and must be checked by a reasoner to see if it is satisfied by the ontology $\mathcal{O}$. We consider trivial statements to be of the form $\mathsf{A} \sqsubseteq \mathsf{A}$ or $\mathsf{A} \sqsubseteq \top$; these will always be satisfied by $\mathcal{O}$.

In order to avoid trivial statements of the form $\mathsf{A} \sqsubseteq \top$, we ignore all variables v in Γ_{q_T} such that $\Gamma_{q_T}(v) = \top$.

In order to avoid trivial statements of the form $\mathsf{A} \sqsubseteq \mathsf{A}$, we ignore all variables v in Γ_{q_T} such that $\Gamma_{q_T}(v) = \Gamma_{q_S}(v)$. Equality is determined using string matching. Note that it is possible for concepts to be ordered differently; this should be accounted for when string matching. For example, $\mathsf{A} \sqcap \mathsf{B}$ is equivalent to $\mathsf{B} \sqcap \mathsf{A}$

This leaves us with non-trivial statements of the form $\Gamma_{q_S}(v) \sqsubseteq \Gamma_{q_T}(v)$, where $\Gamma_{q_T}(v) \neq \Gamma_{q_S}(v)$ and $\Gamma_{q_T}(v) \neq \top$. Such a statement asserts that the concept associated with v in q_T subsumes the concept associated with v in q_S. If $\mathcal{O}$ satisfies this statement, then v in q is consistent with the semantics of S and T with respect to $\mathcal{O}$.

The satisfiability of non-trivial statements must be checked using a reasoner; thus the non-trivial statements are compiled into a list that is output by Algorithm 2.

5.2.2 Finding Associated Concepts

Finding the concepts associated with a variable is a key part of our framework. Associated concepts are used to populate the maps Γ_{q_S} and Γ_{q_T} in Algorithm 1, which are then used to generate the statements about the implied meanings of queries that are checked by a reasoner to perform semantic verification.

Concepts associated with a variable in a query differ from the semantic annotation mappings in that semantic annotation mappings map specific attributes of relations to concepts, whereas concepts associated with a variable take into account all occurrences of the same variable throughout multiple relations in a query. Algorithm 3 uses the semantic annotation mapping to find the concepts associated with a variable in a query.

Algorithm 3 takes as input: the variable v for which an associated concept is to be found; a conjunctive query q_X, which is a conjunction of relations, or *subgoals*, in a schema X (in the context of our main algorithm, Algorithm 1, this will be either q_T or q_S from the query $q : q_T \leftarrow q_S$); and the semantic annotation of schema X, α^X. The algorithm returns $\Gamma_{q_X}(v)$, the concept associated with v in q_X, based on the semantic annotation α^X.

We begin by setting $\Gamma_{q_X}(v)$ to be the universal concept $\top$. This is the default *meaning* of a

Algorithm 3: FindAssociatedConceptsInQuery

input: Variable v, conjunctive query q_X, and semantic annotation α^X.
output: $\Gamma_{q_X}(v)$, the concept associated with v in q_X

$\Gamma_{q_X}(v) \coloneqq \top$
foreach *subgoal* $r(\mathbf{v}) \in q_X$ **do**
  Let $j_1, ..., j_k$ be the positions where v appears in $r(\mathbf{v})$
  foreach $j \in j_1, ..., j_k$ **do**
    $\Gamma_{q_X}(v) \coloneqq \Gamma_{q_X}(v) \sqcap \alpha_r(j)$
  end
  if $\Delta_r(v) \neq \emptyset$ **then**
    foreach $u \in \Delta_r(v)$ **do**
      $\Gamma_{q_X}(v) = \Gamma_{q_X}(v) \sqcap$ FindAssociatedConceptsInQuery (u, q_X, α^X)
    end
  end
end
return $\Gamma_{q_X}(v)$

variable if α^X does not assign a concept to it. Note that $\top \sqcap \mathsf{A} \equiv \mathsf{A}$, since $\mathsf{A} \sqsubseteq \top$ for all concepts A in an ontology.

The algorithm iterates over each subgoal $r(\mathbf{v})$ in q_X[2]. For each subgoal, it does two things: one is to find the concepts associated with v in each position in r in which v appears ($\gamma_r(v, j)$); the other is to find the concepts associated with the variables in the key constraint set of v in r, $\Delta_r(v)$, using a recursive call.

The end result, $\Gamma_{q_X}(v)$, is a conjunction of all concepts associated with v in all positions and in all relations in which v occurs, and all concepts associated with $\Delta_r(v)$ for all relations r in which v occurs.

Iterating Over Positions of r

When we iterate over the positions it is because the semantic annotation mappings assign concepts to individual positions of relations. At each position j we find $\alpha_r(j)$, the concept mapped to position j of the subgoal, and add it to the existing concept associated with v with the $\sqcap$ operator.

The Key Constraint Set of v

The next step is to find the concepts associated with all of the variables in the key constraint set of v. This occurs when a_j is part of a relation that has a primary key, and is especially important when the variable representing that primary key occurs in other relations in q_X.

To illustrate, we return to an example that was introduced in Section 4.3.1. Suppose we have input into Algorithm 3 the following query from our running example:

$$q_S : S.\texttt{Project}(x_1, y, -), S.\texttt{Employee}(y, x_2, x_3)$$

[2]Note that if we have a query $q : q_T \leftarrow q_S$ and pass q_T as an argument, there will only be one subgoal.

And suppose we have these semantic annotation mappings for S:

$\alpha_{\texttt{Project}} : \{2 \rightarrow \mathsf{TechStaff}, 3 \rightarrow \mathsf{AdminStaff}\}$
$\alpha_{\texttt{Employee}} : \{1 \rightarrow \mathsf{Employee}, 2 \rightarrow \mathsf{Employee}, 3 \rightarrow \mathsf{Employee}\}$

Now suppose we want to find the concept associated with the variable x_2 in q_S. For the subgoal `Employee`, the loop on positions would find that $\Gamma_{q_X}(x_2) = \mathsf{Employee}$. This is because the attribute at position 2 in `Employee` is annotated with the concept Employee using $\alpha_{\texttt{Employee}}$.

However, the variable y is in the key constraint set of x_2, $\Delta_{\texttt{Employee}}(x_2)$; it is the primary key for `Employee`. In addition, y occurs in another relation in q_S, `Project`, so that q_S joins the attribute at position 2 in `Project`, `techlead`, with the `Employee` relation. Essentially, q_S is saying that `Project.techlead` refers directly to the subgoal `Employee` with y as its primary key. Thus the meaning of x_2 is dependent upon the meaning of `Project.techlead`; in fact, its meaning is dependent upon the meanings of all attributes represented by y in q_S.

Thus Algorithm 3 finds the concepts associated with y (the only member of $\Delta_{\texttt{Employee}}(x_2)$) and appends them to $\Gamma_{q_S}(x_2)$ using the $\sqcap$ operator.

The result is that $\Gamma_{q_S}(x_2)$ = Employee $\sqcap$ TechStaff.

5.3 Obtaining Semantic Annotations

Semantic annotations can be seen as correspondences between individual attributes in a schema and concepts in an ontology. While they could be chosen manually by a schema mapping designer, we envision them being derived from a set of *alignments* created semi-automatically by a tool such as MapOnto [An et al., 2005].

In MapOnto, an alignment between a relational schema and an ontology is a set of local-as-view (LAV) mappings of the form $r(\mathbf{x}) \rightarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$, where r is a relation, $\mathbf{x}$ is the set of attributes in r, y is a set of variables, and is a conjunctive formula over the ontology.

For example, possible alignments for the source and target schemas S and T in our running example are:

$S.\texttt{Project}(name, techlead, manager) \rightarrow \exists p$, $\mathcal{O}$:Project(p),
$\mathcal{O}$:hasName(p, name),
$\mathcal{O}$:TechStaff(techlead),
$\mathcal{O}$:hasLeader(p, techlead),
$\mathcal{O}$:AdminStaff(manager),
$\mathcal{O}$:hasManager(p, manager)

$S.\texttt{Employee}(eid, name, email) \rightarrow \mathcal{O}$:Employee(eid),
$\mathcal{O}$:hasName(eid, name),
$\mathcal{O}$:hasEmail(eid, email)

$T.\texttt{Program}(pname, supervisor, contact) \rightarrow \exists p, e$, $\mathcal{O}$:Project(p),
$\mathcal{O}$:AdminStaff(e),
$\mathcal{O}$:hasName(p, pname),
$\mathcal{O}$:hasName(e, supervisor),
$\mathcal{O}$:hasEmail(e, contact)

Algorithm 4: AlignmentToSemanticAnnotation

input: Alignment $r(\mathbf{x}) \rightarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$
output: Semantic annotation α_r

local map α_r

Let n be the number of attributes in $\mathbf{x}$.
foreach $i : 1..n$ **do**
  Let $x_i \in \mathbf{x}$ be the attribute at position i in $r(\mathbf{x})$
  $\alpha_r(i) = \top$
  foreach *predicate* $\mathcal{O}$*:*$C(x_i)$ *in* ϕ **do**
    $\alpha_r(i) \coloneqq \alpha_r(i) \sqcap \mathcal{O}$*:*$C$
  end
  foreach *predicate* $\mathcal{O}$*:*$p(u, x_i)$ *in* ϕ*, where* p *is a data property* **do**
    foreach *predicate* $\mathcal{O}$*:*$C(u)$ **do**
      $\alpha_r(i) \coloneqq \alpha_r(i) \sqcap \mathcal{O}$*:*$C$
    end
  end
end
return α_r

Our algorithm for transforming alignments into semantic annotations is Algorithm 4. This algorithm takes as input an alignment for a relation r and returns a semantic annotation for that relation, α_r.

Algorithm 4 iterates over each attribute x in r and searches for predicates describing x on the right hand side of the alignment. For those predicates that describe x as a concept (e.g. TechStaff(x)), the concept is added to the contents of α_r using a conjunction. For those predicates that describe x as the value of a data property for some concept (e.g. hasName(e, x)), the concepts associated with the variable in the domain of that data property are added to the contents of α_r using a conjunction.

Note that it does not make sense for an attribute to be described as both the value of a data property and a concept itself (although it can be the value of an object property and a concept)

For example, the alignment for S.`Project` shown above would result in the following semantic annotation:

$$\alpha_{\texttt{Project}}\colon \{1 \rightarrow \mathsf{Project}, 2 \rightarrow \mathsf{TechStaff}, 3 \rightarrow \mathsf{AdminStaff}\}$$

The first attribute, `name`, becomes annotated with the concept Project because in the alignment we have the predicate $\mathcal{O}$:hasName(p, `name`), which states that `name` is the value of the data property hasName for p; furthermore, based on the predicate $\mathcal{O}$:Project(p) in the alignment, we can say that p is annotated with the concept Project.

The second attribute, `techlead`, becomes annotated with the concept TechStaff because in the alignment we have the predicate $\mathcal{O}$:TechStaff(`techlead`). Although `techlead` also occurs in another predicate, $\mathcal{O}$:hasLeader(p, `techlead`), we disregard this predicate because it describes an *object* property rather than a *data* property.

Similarly, the second attribute, `manager`, becomes annotated with the concept AdminStaff because in the alignment we have the predicate $\mathcal{O}$:AdminStaff(`manager`).

5.4 Conclusion

This chapter explained the framework we use for mapping verification, which involves finding the concepts associated with attributes in source and target schemas based on semantic annotation, rewriting queries in a mapping as statements in terms of these associated concepts, then using a reasoner to check these statements for satisfiability by the ontology used. We presented two main algorithms: one for verifying a mapping, and one for finding concepts associated with variables in a query based on a semantic annotation.

The next chapter provides a discussion of this framework and the algorithms.

Chapter 6

Discussion

This chapter provides a discussion of the feasibility of using our framework as a solution to the mapping verification problem.

6.1 Context

This framework was designed in the context of the Semantic Web and *critical* data integration settings.

This framework was designed with the Semantic Web in mind. That is, the degree of its feasibility is partially dependent upon the availability of certain Semantic Web technologies: domain-specific ontologies and reasoners. It is still unclear to what degree this vision will become a reality, but the Semantic Web research community is an active one. However, both ontologies and reasoners are already well-defined and available in mature forms, though they are not in widespread use.

In addition, our framework is meant for *critical* data exchange settings rather than *casual* ones. In the casual setting, the steps needed to obtain the inputs to our framework - finding a suitable ontology, creating semantic annotations, and creating proposed schema mappings - would likely take longer than mapping a small database schema by hand. In addition, in such settings, failure to map the data correctly is not catastrophic, so mapping verification is not as important.

However, in the critical setting, we argue that the added cost of using our framework - finding an ontology and creating semantic annotations - is not unreasonable, particularly in a scenario in which ontologies are widely available and the same schemas are likely to be involved in data exchange more than once. In such a setting it is expected that the data exchange will take a long time to complete, that a schema mapping generation tool such as Clio [Miller et al., 2001b] will be used, and that the data will be mapped accurately.

It is with this critical data integration scenario in mind that we frame the following discussion.

6.2 Ontologies and Semantic Annotations

Ontologies were a natural choice for our framework because, in the context of the Semantic Web, they are being used to provide a shared vocabulary for heterogeneous data sources. Using ontologies and semantic annotations, our solution provides a formal way to express the semantics of schemas so that they are both machine-readable and human-understandable.

6.2.1 Advantages of Ontologies

With semantic annotations, ontologies become a single point of reference for the vocabulary of the domain of the source and target schemas. In the context of our framework, this means that there is no longer any need for the schema mapping designer to be an expert on both schemas. Normally, when the schema mapping designer is verifying schema mappings by hand, he must first take the time to understand both schemas in detail. Using our framework, the semantic annotation and the ontology provide a translation of the schema elements in terms of the vocabulary of the ontology. Thus, not only is the bulk of the mapping verification done automatically, but understanding the schemas is also made easier.

There are several advantages to using ontologies to provide a terminology for multiple relational schemas.

First, the language ontologies are expressed in is understandable to both humans and machines. Ontologies are expressed formally in a language suitable for automatic reasoning. However, this formal language does not preclude the use of an arbitrary natural vocabulary to describe concepts and relationships, making ontologies understandable to humans as well (for example, statements that can be formally expressed in an ontology include TechnicalStaff isA Employee, or Employee hasName personName). This is in contrast to relational schemas, whose attributes and relations are often given short names that are not meaningful. Relational schemas are also generally normalized for query efficiency, which makes them even less readable to humans.

Another advantage of ontologies is that their level of expressiveness is high enough that they have great potential for reuse in various applications and over many different schemas. Ontologies can define detailed classes of objects and rich relationships between these classes. They are more expressive than relational schemas; therefore, we can expect an ontology in a particular domain of discourse to be able to describe several schemas in that domain (relational or otherwise). The implication of this is that, in a scenario where our mapping verification framework is used on two schemas, the semantic annotations could be useful in subsequent operations involving these and other schemas.

Ontologies can capture the subtleties of the domain they are describing, as understood by the ontology designer. They are unambiguous - generally, one concept is used per real-world entity, but equivalence of concepts can also be specified. This, combined with their high expressiveness, means that ontologies can be used in order to become familiar with a new domain of discourse,

for instance when a new employee is hired or when a schema mapping designer is working with unfamiliar schemas.

6.2.2 Entity-Relationship Diagrams

It could be argued that entity-relationship (ER) diagrams would be a suitable way of annotating the semantics of relational schemas, since, like ontologies, they are conceptual models of the schemas. The advantage of ER diagrams is that they are usually created during the schema design process, and so they are readily available. However, there are several reasons why ER diagrams are not suitable for our framework.

ER diagrams are not as expressive enough to allow the specification of complex domain constraints that may be of interest when trying to understand a domain of discourse (e.g. "an Experienced Manager is one who has managed at least one project"). Additionally, ER diagrams do not reflect the changes in the schemas after schema normalization is performed. ER diagrams are also very specific to one schema and may not be general enough to model a different schema.

6.2.3 Disadvantages of Ontologies

The high expressivity of ontologies is advantageous, but has some downsides. Large ontologies can be overwhelming to try to understand without a good visualization tool. In addition, the paradigm used in ontologies is subtly different from that used in relational databases, in that ontologies are designed using the open world assumption, whereas relational databases use the closed world assumption. (This is explained further in Section 6.3). There is also a tradeoff between expressiveness of an ontology and tractability of reasoning, which is explained in Section 6.4.

In the context of the Semantic Web, one problem that occurs with ontologies is heterogeneity of ontologies. This somewhat hampers the advantage of reuse of semantic annotations. Suppose we have two schemas, each annotated to a different ontology in the same domain. We must now decide whether to create new annotations for one of the schemas, or try to align the two ontologies to each other. However, it must be noted that ontology alignment is also an active research topic.

6.2.4 Assumptions about ontologies

We make a number of assumptions about the robustness and specificity of the ontologies used in our framework.

First of all, we assume that the ontology we use is robust enough to encompass all significant[1] schema elements in both schemas. Such a domain ontology could be available from a third-party source, however it is possible that even a robust third-party ontology would have to be modified to fit the needs of the tool. In the absence of an adequate third-party ontology, the schema designers would have to create the ontology themselves - a time-consuming undertaking. However, there are

[1] Generally, the significant schema elements are those to and from which data is moved, especially those elements for which accuracy in the schema mapping is critical.

tools for ontology creation (such as Protege[2]) that would make this process easier. In addition, as we have already discussed, ontologies have the potential for reuse, so creating an ontology could benefit more than just the task of finding schema mappings.

Secondly, we assume that both the ontology and the semantic annotations are specific enough to accurately describe the meaning of the schema elements. For instance, in our running example, if the ontology did not make a distinction between AdminStaff and TechStaff and instead had only the concept Employee, the ontology would not be specific enough to distinguish between the `manager` and `techlead` attributes in the source schema. On the other hand, if both `manager` and `techlead` were annotated with Employee, then it would be assumed that there is no relevant distinction between their meanings. If in fact there *were* a relevant distinction between the meanings of `manager` and `techlead`, but they were still annotated with Employee, then this semantic annotation would not be specific enough to distinguish between them (i.e it would be incorrect). We assume that the schema mapping designer would be aware of this fact and be skilled enough to provide as much information about each schema element as is necessary despite the fact that semantic annotations are only required to be partial mappings.

6.3 Reasoners

Typically applications that use reasoners for tasks in the realm of databases must deal with a subtle mismatch between the assumptions behind reasoners and databases; our framework avoids these problems by using reasoners as they were intended to be used: as automatic type classifiers.

Contrary to the typical behaviour of tools designed for checking consistency (such as syntax checkers or compilers), reasoners are not designed for the purpose of debugging. Instead, they are designed to (a) determine if a particular statement is entailed by an ontology; and (b) infer new statements, or facts, based on existing facts in the ontology. This results in some subtle behaviours that are unintuitive or unexpected for those with a database background, as many schema mapping designers are.

This problem stems largely from a mismatch in the assumptions behind knowledge bases and databases. Databases are created using a closed-world-assumption, which means that in practise information in the database is generally complete. Unknown information can be denoted by using null values, but it is also possible to specify that certain fields must be complete (i.e. not null). For instance, if we assert that every employee has a Social Insurance Number (SIN), we can have a constraint in the database that every tuple in the `Employee` relation must have a non-null value for the `SIN` field. In practise, if we create a tuple in the database describing the employee, the database will *require* that we input a non-null value for his or her SIN. Ontologies, on the other hand, use the open-world assumption. If there is an assertion in the TBox of an ontology that every Employee has a SIN, and we create an instance of Employee, it is not actually *necessary* to associate a SIN

[2]http://protege.stanford.edu

with that instance. Instead, a reasoner will assume that because every Employee has a SIN, this particular instance of Employee has a SIN, even if this SIN is not specified.

In short, there is no way to specify a database-style integrity constraint in an ontology. There is also no way to specify type constraints.

This mismatch problem is described in detail by Motik et al. [Motik et al., 2007]. Here the authors cite the example of BioPAX[3], an ontology used for exchanging biological pathway data. BioPAX defines a property NAME as follows:

$$\exists \mathsf{NAME}.\top \sqsubseteq \mathsf{bioSource} \sqcup \mathsf{entity} \sqcup \mathsf{dataSource}$$

The intent of this property is to constrain the types of objects that can be named; that is, to ensure that only objects of type bioSource, entity or dataSource can have a name. However, according to this property, when a reasoner encounters an object with a name it will *infer* that the object is of type bioSource, entity or dataSource rather than *checking* that this is so.

Such subtle differences must be understood when working with relational databases and ontologies.

It is important to note that this mismatch problem occurs only when working with both the TBox and the ABox of the ontology; that is, it only becomes an issue when working with both the terminology and the individual instances of concepts, where there might be some overlap between the role of a database (storing data) and the ontology (storing information about individuals). This is not a problem we are concerned with because our framework works only with the TBox of an ontology.

When working with the reasoner, our framework is mainly concerned with the question: can the description of one attribute fill the role of the description of another attribute? Essentially, it is asking the reasoner to perform *type* checking rather than constraint checking. Because reasoners perform automatic type classification, they are well-suited to this task.

6.4 Computational Complexity

The running time of our framework is largely dependent upon the size and complexity of inputs into the reasoner. Here we describe the complexity of finding these inputs and discuss the performance of the reasoner.

6.4.1 The Cost of Finding Associated Concepts

Recall that, given a query q_X, a semantic annotation α^X, and a variable v, Algorithm 3 finds the concept associated with v in q_X, $\Gamma_{q_X}(v)$. This is used later in our framework to generate the verification statements that are input to the reasoner.

[3] http://www.biopax.org

Note that given a variable v, Algorithm 3 iterates on the subgoals of q_X, and at each subgoal r, determines whether to make a recursive call based on the contents of $\Delta_r(v)$. Thus in order for Algorithm 3 to terminate, we must assume that it is *cycle-free*, that is:

DEFINITION 13 (CYCLE-FREE) *Let $q = r_1(\mathbf{v}_1), ..., r_k(\mathbf{v}_k)$ be a conjunctive query in which distinct variables v, u appear. q is* cycle-free *if there are no subgoals r_i, r_j in q such that $u \in \Delta_{r_i}(v)$ and $v \in \Delta_{r_j}(u)$.*

Let us assume that Algorithm 3 is given q_X, α^X, v, as well as $\Delta_r(v)$ for each relation r and variable v in q_X, and that α^X and $\Delta_r(v)$ can be accessed in constant time.

Then Algorithm 3 computes $\Gamma_{q_X}(v)$ in $\mathcal{O}(v^2|q_x|)$ time, where v is the maximum number of variables in any subgoal of q_X, and $|q_x|$ is the total number of subgoals in q_X.

6.4.2 The Cost of Finding Verification Statements

Verification statements are computed by Algorithm 2. Given a query $q : q_T \leftarrow q_S$, Algorithm 2 takes as input Γ_{q_S} and Γ_{q_T}, and generates the verification statements in $\mathcal{O}(|\Gamma_{q_T}|)$ time, where $|\Gamma_{q_T}|$ is the number of variables in Γ_{q_T}. The number of statements generated is also in $\mathcal{O}(|\Gamma_{q_T}|)$.

Our main mapping verification algorithm, Algorithm 1, generates and checks verification statements for each query in a given mapping $\mu : S \rightarrow T$. The total number of verification statements checked by the reasoner is in $\mathcal{O}(m|\mu|)$, where $|\mu|$ is the number of queries $q : q_T \leftarrow q_S$ in the mapping to be verified, and m is the maximum number of variables in Γ_{q_T} for all q_T in any query in μ.

Thus we see that the number of inputs to the reasoner - the verification statements - is polynomial on the size of the input.

In addition, the maximum number of terms in a verification statement is polynomial in the maximum number of subgoals among queries in μ, the maximum number of variables among them, and the maximum number of predicates among semantic annotation mappings in the semantic annotations for the source and target schemas.

Knowing that the number of verification statements and the length of these statements is bounded by polynomials on the size of the inputs to the main algorithm, what remains is to determine the cost of checking the consistency of these verification statements with the ontology.

6.4.3 The Cost of Reasoning

In general, the computational cost of reasoning over an ontology depends upon the expressiveness of the language used to represent the ontology. Reasoning on full first-order logic (FOL) is intractable, however there are subsets of FOL on which reasoning is tractable.

We have based our framework upon ontologies expressed in OWL-DL, which is known to correspond to the descrption logic language $\mathcal{SHOIN}(\mathbf{D})$. Reasoning on $\mathcal{SHOIN}(\mathbf{D})$ has been found

to be NEXPTIME-complete [Tobies, 2001].

However, in practise, reasoners are optimized to perform quickly for the most common queries. For instance, Pellet [Sirin et al., 2007] is a reasoner that is optimized to work with OWL-DL ontologies on the the Semantic Web. FaCT++ [Tsarkov and Horrocks, 2006] is another optimized reasoner. Optimizations can make a significant difference in reasoner performance; for example, Horrocks [Horrocks, 1998] showed that the FaCT system (a reasoner supporting the $\mathcal{ALCH}_{R^+}$ description logic) was able to classify a large, realistic knowledge base in 379s of CPU time, for an average of 0.003s per subsumption test for 122,695 tests. It took 100 hours of CPU time for an unoptimized reasoner (KRIS [Baader and Hollunder, 1991]) to classify just 10% of a simplified version of this knowledge base.

Thus, in practise, we expect that the reasoner will terminate in a reasonable amount of time for most statements. In an implementation of our framework, it is not unreasonable to terminate the reasoner if it does not finish evaluating a statement after a certain number of steps; for these statements, verification can be done by hand.

6.5 Implementation

Our framework is conceptual, but it is possible to build a working tool based upon it. It is designed to complement existing tools related to data integration (schema mapping creation tools) as well as mature Semantic Web technologies (ontologies and reasoners).

We envision a tool based on our framework being used alongside an existing tool for finding schema mappings, such as Clio [Miller et al., 2001b]. Clio or a similar tool would provide one of the inputs - the executable schema mapping to be verified.

Our tool could be written in three main modules: finding associated concepts, generating verification statements, and reasoning.

The semantic annotations could be created by hand or in conjunction with tools for finding semantic annotations automatically, such as MapOnto [An et al., 2006]. Alternatively, the module for finding associated concepts could be ignored, and associated concepts could be defined by hand. However, such a solution would make it difficult to reuse these associated concepts later, since associated concepts are much more general than semantic annotations.

We have assumed in this framework that the ontology would be specified in OWL-DL. It is certainly possible for other languages to be used as well, as long as a compatible reasoner is provided. However, this would have implications on what kind of information could be expressed in the semantic annotations as well as the running time of reasoning.

Given that the ontology is specified in OWL-DL, we recommend that a reasoner such as Pellet [Parsia and Sirin, 2004] or FaCT++ [Tsarkov and Horrocks, 2006] is used.

6.6 Feasibility of Use

The initial motivation for creating this framework was to find a way to reduce the cost and difficulty of performing mapping verification as compared to verifying mappings by hand.

Although a full evaluation of a working tool would be needed to determine whether the framework truly reduces the cost of mapping verification, we outline here a number of expected benefits and downsides.

6.6.1 Time Investment

Finding schema mappings is already a time-consuming process; this is unavoidable. The question is whether using our framework would reduce the time needed to verify mappings as opposed to doing so by hand, which is the predominant method used today. Note that some user involvement is inevitable in a critical data integration scenario, as it is unreasonable to expect our framework to be accurate in all mapping verification cases; we hope and expect to minimize this user involvement.

Our framework requires some initial time investment to find a suitable ontology and create semantic annotations. Quality ontologies are difficult to find today, but in the context of a more advanced Semantic Web, such ontologies should be much more widely available. In fact, it is not unreasonable to expect that many schemas will already be annotated with ontologies at design-time due to the potential applications in data interoperability.

Creating semantic annotations could be time consuming for large schemas, but recall that there is already some research in automating the task of creating detailed alignments between schemas and ontologies, such as the MapOnto tool [An et al., 2006]. As we described in Section 5.3, it is possible to turn these automatically create alignments into the less specific semantic annotations needed for our framework by using Algorithm 4.

Note also that we do not require semantic annotations to be complete mappings, so the time needed to create them manually could be reduced by annotating only a subset of the schemas. However, the accuracy of mapping verification is dependent on how accurate and specific the given semantic annotations are.

An additional cost is that of maintaining the semantic annotations over the long-term. If the semantic annotations and ontology are not to be reused, this is not a problem. However, we have argued that there are many hidden potential benefits to having semantically annotated schemas, both for the purposes of data interoperability and of documentation. Thus maintenance of the semantic annotations would be necessary.

We believe that, on the whole, our semi-automatic reasoning-based framework would be faster than manual mapping verification for large-scale critical data exchange tasks. However, without an evaluation it is impossible to tell how quickly an automatic reasoner could compute the entailment of the most common types of verification statements. If reasoning is fast for the majority of verification

statements, it is possible to implement the tool in such a way that, for statements that take unusually long to verify, the reasoner is terminated and the user performs the verification.

6.6.2 Evaluation

The true costs and benefits of this framework cannot be known without doing a thorough evaluation. Such an evaluation would be a large undertaking due to our expectation that this would be most useful in a critical data integration setting. We discuss this further in Chapter 7.

6.7 Future Possibilities

This framework could serve as the basis of many future enhancements and directions.

As Semantic Web research advances, ways to improve our framework might emerge - for instance, there may be reasoners designed to work better in the context of relational database schemas. In addition, some advances in ontology alignment (the equivalent of schema mapping for ontologies) might be applicable to our framework.

Since the Semantic Web is focused on data interoperability of heterogeneous schemas, it is also conceivable that we could extend this framework to work with other kinds of schemas. This is discussed further in Chapter 7.

6.8 Conclusion

In this chapter we have discussed how our conceptual framework could make the mapping verification step of designing schema mappings more cost- and time-efficient as compared to verifying mappings by hand in a critical data exchange setting. We showed that the benefits of this framework would be strongest in the context of a flourishing Semantic Web with abundant ontologies, reasoners, and many applications that serve as incentives for semantically annotating schemas. We also noted the most significant concern of our framework - whether the time to semantically annotate a schema and the computational demands of reasoners outweighs the time it takes to verify schema mappings by hand.

In the next chapter we discuss the potential future directions we could take with this framework.

Chapter 7

Future Work

The framework we have presented lays the foundation for a number of possible future directions. We divide these into two categories: those that are directly related to mapping verification, and those that extend the functionality of the framework beyond mapping verification.

7.1 Future Work in Mapping Verification

For the framework in its current state, we would like to perform a comprehensive evaluation of its practical effectiveness, display explanations for inconsistencies, and see it extended to work with other kinds of schemas.

7.1.1 Explaining Semantic Inconsistency Using Proof Trees

Reasoners can provide *proof trees*, which show the steps that were used to arrive at a conclusion about whether a statement is satisfied by an ontology. Given such proof trees, it is possible to retrace these steps and provide an explanation of why an inconsistent statement is deemed to be so.

Such explanations would be useful to a schema mapping designer as they would pinpoint the source of the inconsistency. The schema mapping designer could use the information to change the semantic annotation or change the schema mapping.

These proof trees could also be useful for automatically suggesting schema mappings, as explained in Section 7.2.1.

7.1.2 Evaluation

An evaluation of this framework is necessary to test two major areas: the precision and recall of testing for semantic consistency, and the practical benefits of the framework in terms of usability, and time saved.

Precision and Recall - Benchmarks

For a meaningful evaluation of precision and recall, we would need a set of source and target schemas in a domain D, a set of ontologies in domain D, a set of semantic annotations between the schemas and ontologies, and a set of correct and incorrect mappings between the source and target schemas. Ideally, this set would contain multiple schemas, ontologies, semantic annotations, and mappings from various domains.

We have found no standardized benchmark for testing the precision and recall of schema mappings, though one notable effort is the Amalgam test suite [Miller et al., 2001a]. Amalgam contains several small schemas and their conceptual models, but does not have full ontologies, semantic annotations, or schema mappings. It would be a worthwhile endeavour to create such a benchmark.

Given a benchmark, it would be possible to automate the testing of precision and recall and measure things such as practical efficiency of the algorithms as well.

Practical Usability

It would also be worthwhile to test the practical usability of this framework. Assuming that ontologies are widely available, does the framework make the job of mapping verification easier and faster than doing mapping verification by hand? How does it compare against statistical mapping verification methods such as the Spicy method [Bonifati et al., 2008].

A meaningful evaluation of practical usability would be quite resource- and time-consuming. Our framework is not designed or expected to be practical for use with small schemas, so larger, more complex schemas would have to be used for the evaluation. While these would not need to be quite as large as the schemas used in real data integration scenarios, they should at least approximate these schemas. Since the process of finding schema mappings is inherently time-consuming, we would expect an evaluation of the usability of the framework to require a large time investment.

7.1.3 Beyond Relational Schemas

As noted earlier, while we frame both the mapping verification problem and our solution in terms of relational databases, the same problem could be applied to other types of schemas, such as XML schemas or natural language schemas (like those on web forms). This could possibly be done by generalizing the internal schema model to work with both structured and semi-structured schema types, as Clio [Popa et al., 2002] does to support many schema types.

7.2 Beyond Mapping Verification

Beyond simply verifying mappings, our framework could be used for suggesting both schema mappings and semantic annotation mappings.

7.2.1 Suggesting Schema Mappings

Aside from simply verifying mappings, the rich domain information present in ontologies combined with semantic annotations could allow us to *suggest* mappings as well.

Given a semantically annotated attribute a in a schema, the ontology can provide information such as which concepts are related to a, and by what kinds of properties. A semantic annotation for a different schema provides clues as to which other attributes a could be mapped to. For instance, a could be mapped into any attribute that is semantically annotated with the same concept as a or its parent concepts. It may be closely related to attributes mapped to concepts related to the concept of a by properties that are not disjointness. Such heuristics could be used to create a scoring function for ranking attributes by their likelihood to be a good match for a.

Such a method could be applied to suggest completions for partial schema mappings, or to suggest alternate queries to make a complete schema mapping consistent.

A partial schema mapping is one in which not all attributes in the source schema are mapped to the target schema. The reasons why a partial mapping may exist are similar to the reasons some tools for finding schema mappings suggest multiple schema mappings: ambiguity in the semantics of the schemas makes it difficult to decide which attributes in the target schema are the best match.

7.2.2 Ranking Candidate Schema Mappings

Whether working with schema mappings suggested by a schema mapping generation tool or schema mappings suggested by a tool based on our framework, it would be useful to have a mechanism for ranking candidate schema mappings according to their likelihood of being the "best" mapping. This could be useful when a number of mappings are found to be semantically consistent and the schema mapping designer must choose the best one. It could also be useful in a situation where *no* mappings are semantically consistent and the schema mapping designer wishes to find the mapping that is closest to being correct to minimize the number of adjustments that must be made manually to the mapping.

Ranking mappings would require quantifying their quality in some way. This might involve using some sort of semantic similarity measure on the ontology in order to determine how "close" the mapping is to being correct, as well as how specific it is. For instance, going back to our running example, if one mapping maps a project in the source schema to a manager in the target schema, and another mapping maps a technical leader in the source schema to a manager in the target schema, both mappings would be semantically inconsistent, but the second mapping would be much closer to being the correct mapping because technical leaders and managers are more closely related semantically than projects and managers.

7.2.3 Suggesting Semantic Annotation Mappings

It is possible that the semantic annotation mappings that are input to the framework will not always be correct, or they might occasionally be too general. This would be a more prominent issue if a tool (such as MapOnto [An et al., 2005]) is used to automatically find semantic annotations.

Thus when our framework discovers a semantically inconsistent query, it may prove useful to be able to suggest alternate semantic annotation mappings that would make the query semantically consistent. Such a feature might use proof trees to pinpoint why the query is semantically inconsistent, then use a combination of reasoning and examining the related concepts and properties to make a suggestion to the schema mapping designer.

Such functionality would be especially useful were semantic annotations to become a way of documenting schemas.

7.3 Conclusion

Our framework opens up a number of research possibilities. The framework as it is could be enhanced by expanding it to work with more types of schemas besides relational schemas, its usability could be enhanced by giving it the ability to explain semantic inconsistencies using proof trees provided by the reasoner, and a thorough evaluation needs to be done to determine the true costs and benefits of using the framework.

Beyond simple mapping verification, there are numerous directions for which this framework lays a foundation. It could be used to not only verify, but also suggest schema mappings, rank schema mappings based on a measure of quality, and suggest semantic annotation mappings as well.

Chapter 8

Conclusion

The major difficulty in attaining data interoperability is the inevitable semantic and syntactic heterogeneity of schemas. Schema mappings are key to overcoming this heterogeneity. In the data exchange approach to data interoperability, the most under-explored area has been the step in which schema mappings are checked for semantic consistency - the mapping verification step. Relatively little research has been dedicated to semi-automatically verifying schema mappings, and most schema mapping creation tools leave the bulk of this step up to the schema mapping designer, who has the additional knowledge about the schemas and the capability to reason about this knowledge needed to decide whether the semantics of a schema mapping are correct. Those approaches to schema mapping verification which do exist generally rely on database instances to provide some additional knowledge about the schemas being mapped, but the knowledge contained in database instances is limited and of inconsistent quality.

The maturation of a few key Semantic Web technologies - ontologies and reasoners - makes it feasible to formally express and reason about external information about schemas without relying on database instances.

We have presented a framework that brings these Semantic Web innovations to data exchange. By annotating schema elements with terms from an ontology, we make available to the schema mapping creation tool a rich layer of background knowledge that cannot be gleaned from database instances alone.

We have defined the notion of *semantic consistency* of a schema mapping in terms of an ontology and presented algorithms that verify this semantic consistency. We annotate the meanings of schema elements using semantic annotations, then use these semantic annotations to rewrite queries in a schema mapping as statements about their meaning in terms of the ontology.

Our framework approaches schema mapping verification from a unique angle and lays the groundwork for a number of exciting possible extensions that could make schema mapping creation a faster, easier, and more automated process.

Bibliography

S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H.G. Molina, D. Gawlick, and Others. The Lowell Database Research Self-Assessment. *Communications of the ACM*, 48:111–118, 2005.

Z. Aleksovski, M. Klein, W. Kate, and F. Van Harmelen. Matching Unstructured Vocabularies Using a Background Ontology. *Lecture Notes in Artificial Intelligence*, 4248:182–197, 2006.

B. Alexe, L. Chiticariu, R.J. Miller, D. Pepper, and W.C. Tan. Muse: A system for understanding and designing mappings. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, pages 1281–1284, Vancouver, BC, Canada, 2008. ACM.

Y. An, A. Borgida, and J. Mylopoulos. Inferring complex semantic mappings between relational tables and ontologies from simple correspondences. *Lecture notes in computer science*, 3761: 1152–1169, 2005.

Y. An, J. Mylopoulos, and A. Borgida. Building semantic mappings from databases to ontologies. *Proceedings of the National Conference on Artificial Intelligence*, 21:1557 – 1560, 2006.

G Antoniou and F van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

D. Aumueller, H.H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with COMA++. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)*, volume pages, pages 906 – 908, Baltimore, Maryland, USA, 2005. ACM.

F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithms. In *Processing declarative knowledge: International workshop PDK'91*, volume 567 of *Lecture Notes in Artificial Intelligence*, pages 67–86. Springer-Verlag, 1991.

F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. *Lecture Notes in Artificial Intelligence*, 2605:228–248, 2005.

F Baader, D Calvanese, DL McGuinness, D Nardi, and P Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2007.

C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, December 1986.

P.A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, volume pages, pages 1 – 12, Beijing, China, 2007. ACM.

A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema Mapping Verification: The Spicy Way. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 85–96. ACM, 2008.

P. Bouquet, L. Serafini, and S. Zanobini. Semantic coordination: a new approach and an application. *Lecture Notes in Computer Science*, 2870:130–145, 2003.

R.J. Brachman and H.J. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann, Jan 2004.

D. Calvanese and G. De Giacomo. Data integration: A logic-based perspective. *AI Magazine*, 26: 59, 2005.

N Choi, IY Song, and H Han. A survey on ontology mapping. *ACM Sigmod Record*, 35:34–41, 2006.

W.W. Cohen, P. Ravikumar, and S.E. Fienberg. A comparison of string distance metrics for name-matching tasks. *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, 2003.

R. Davis, H. Shrobe, and P. Szolovits. What is a knowledge representation? *AI magazine*, 14:17, 1993.

A. Doan, J. Madhavan, P. Domingos, and A. Halevy. *Ontology matching: A machine learning approach*, pages 385–516. Springer Verlag, Berlin, Heiderlberg, New York, 2003.

A.H. Doan. Learning to map between structured representations of data, 2002.

A.H. Doan and A.Y. Halevy. Semantic integration research in the database community: A brief survey. *AI magazine*, 26:83, 2004.

A.H. Doan, P. Domingos, and A.Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 509–520, New York, NY, USA, 2001. ACM.

A.H. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to Map between Ontologies on the Semantic Web. In *Proceedings of the 11th international conference on World Wide Web (WWW '02)*, page 673, Honolulu, Hawaii, USA, 2002. ACM.

M. Ehrig and Y. Sure. Ontology mapping-an integrated approach. *Lecture Notes in Computer Science*, 3053:76–91, 2004.

D.W. Embley. Toward semantic understanding: an approach based on information extraction ontologies. In *Proceedings of the 15th Australasian database conference*, pages 3–12, Adelaide, Australia, 2004. Australian Computer Society, Inc.

D.W. Embley, L. Xu, and Y. Ding. Automatic direct and indirect schema mapping: experiences and lessons learned. *ACM SIGMOD Record*, 33:14–19, 2004.

R Fagin, P G Kolaitis, R J Miller, and L Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005a.

R. Fagin, P.G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Transactions on Database Systems*, 30:174–210, March 2005b.

Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.

Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321193687.

M. Friedman, A. Levy, and T. Millstein. Navigational Plans for Data Integration. *Proceedings of the National Conference on Artificial Intelligence (AAAI '99)*, pages 67–73, 1999.

A Gal. Why is schema matching tough and what can we do about it? *SIGMOD Rec.*, 2006.

H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in TSIMMIS. *Proceedings of the AAAI Symposium on Information Gathering*, 3:61–64, 1995.

F. Giunchiglia and P. Shvaiko. Semantic matching. *The Knowledge Engineering Review*, 18:265–280, September 2004.

V. Haarslev and R. Möller. Racer: An OWL Reasoning Agent for the Semantic Web. In *Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems*, pages 91–95. IEEE Comput. Soc, 2003.

A. Halevy, A. Rajaraman, and J. Ordille. Data integration: The teenage years. In *Proceedings of the 32nd international conference on Very Large Data Bases*, page 16, Seoul, Korea, 2006. ACM.

I. Horrocks. Using an expressive description logic: Fact or fiction? In *Principles of Knowledge Representation and Reasoning*, 1998.

I. Horrocks. DAML+OIL: A Description Logic for the Semantic Web. *Bulletin of the IEEE Computer Society TEchnical Committee on Data Engineering*, pages 1–7, 2007.

I. Horrocks and P.F. Patel-Schneider. Reducing owl entailment to description logic satisfiability. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):345 – 357, 2004. ISSN 1570-8268. International Semantic Web Conference 2003.

D.F. Huynh, R.C. Miller, and D.R. Karger. Potluck: Data mash-up tool for casual users. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):274 – 282, 2008. ISSN 1570-8268.

Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18:1–31, 2003.

V Kashyap, C Bussler, and M Moran. *The Semantic Web*. Springer-Verlag, 2008.

A Kementsietsidis, M Arenas, and RJ Miller. Mapping data in peer-to-peer systems: semantics and algorithmic issues. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003.

Y. Lei. An instance mapping ontology for the semantic web. In *Proceedings of the 3rd international conference on Knowledge capture - K-CAP '05*, pages 67–74, Banff, Alberta, Canada, 2005. ACM Press.

M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM New York, NY, USA, 2002.

A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 251–262. IEEE Comput. Soc, 1996.

Y Li, Z Bandar, and D McLean. An approach for measuring semantic similarity between words using multiple information sources. Jan 2003.

D. Lin. An information-theoretic definition of similarity. In *Proceedings of the 15th International Conference on Machine Learning*, volume 296304, 1998.

J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 49–58, Rome, Italy, 2001.

C. Meilicke, H. Stuckenschmidt, and A. Tamilin. Improving automatically created mappings using logical reasoning. In *ISCW Workshop, Ontology Matching (OM-2006)*, pages 61–72, 2006.

S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. *Proceedings 18th International Conference on Data Engineering*, pages 117–128, 2002.

F. Mesquita, D. Barbosa, E. Cortez, and A.S. Da Silva. FleDEx: flexible data exchange. In *Proceedings of the 9th annual ACM international workshop on Web information and data management (WIDM '07)*, pages 25–32, Lisboa, Portugal, 2007. ACM.

R.J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam Schema and Data Integration Test Suite. www.cs.toronto.edu/ miller/amalgam, 2001a.

R.J. Miller, M.A. Hernández, L.M. Haas, L. Yan, C.T.H. Ho, R. Fagin, and L. Popa. The Clio project: managing heterogeneity. *ACM Sigmod Record*, 30:78–83, 2001b.

M. Minsky. *A framework for representing knowledge.*, pages 95–128. MIT Press, Cambridge, MA, 1981.

B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between OWL and relational databases. In *Proceedings of the 16th international conference on World Wide Web - WWW '07*, pages 807 – 816, Banff, Alberta, Canada, 2007. ACM Press.

I. Niles and A. Pease. Towards a standard upper ontology. *Proceedings of the international conference on Formal Ontology in Information Systems*, 2001:2–9, 2001.

N.F. Noy. Semantic integration: a survey of ontology-based approaches. *ACM Sigmod Record*, 33: 65–70, 2004.

B Parsia and E Sirin. Pellet: An owl dl reasoner. *Third International Semantic Web Conference-Poster*, Jan 2004.

P.F. Patel-Schneider. A decidable first-order logic for knowledge representation. *Journal of Automated Reasoning*, 6:361–388, December 1990.

T. Pedersen, S. Patwardhan, and J. Michelizzi. Wordnet::Similarity - Measuring the relatedness of concepts. *Proceedings of the National Conference on Artificial Intelligence*, 19:1024–1025, 2004.

A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *Lecture Notes in Computer Science*, 4900:133–173, 2008.

L. Popa, Y. Velegrakis, M.A. Hernández, R.J. Miller, and R. Fagin. Translating Web Data. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 609, Hong Kong, China, 2002. ACM.

H. Qin, D. Dou, and P. LePendu. Discovering executable semantic mappings between ontologies. *Lecture Notes in Computer Science*, 4803:832–849, 2007.

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

J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12: 23–41, April 1965.

G Rull, C Farre, E Teniente, and T Urpi. Validation of mappings between schemas. *Data & Knowledge Engineering*, 66:414–437, September 2008.

H.H. Shahri, J. Hendler, and D. Perlis. Grounding the Foundations of Ontology Mapping on the Neglected Interoperability Ambition. *Association for the Advancement of Artificial Intelligence*, 2008.

V. Sikka. Next generation data management in enterprise application platforms. In *Very Large Data BasesProceedings of the 32nd international conference on Very large data bases*. ACM, 2006.

E Sirin, B Parsia, B Grau, a Kalyanpur, and Y Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5:51–53, 2007.

R Smullyan. *First-order logic*. 1995.

S. Tobies. Complexity Results and Practical Algorithms for Logics in Knowledge Representation. *CoRR*, cs.LO/0106031, 2001.

D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. *Lecture Notes in Computer Science*, 4130:292–297, 2006.

D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks. Using Vampire to reason with OWL. *Lecture Notes in Computer Science*, 3298:471–485, 2004.

O. Udrea, L. Getoor, and R.J. Miller. Leveraging data and structure in ontology integration. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, page 460. ACM, 2007.

R.H. Warren and F.W. Tompa. Multi-column substring matching for database schema translation. In *Proceedings of the 32nd international conference on Very large data bases (VLDB '06)*, page 342, Seoul, Korea, 2006. ACM.

L. Wyatt, T. Shea, and D. Powell. We loaded 1tb in 30 minutes with ssis, and so can you. Technical report, Microsoft, March 2009.

A.V. Zhdanova and P. Shvaiko. Community-driven ontology matching. *Lecture Notes in Computer Science*, 4011:34–49, 2006.