The Uni-Level Description: A Uniform Framework for Managing Structural Heterogeneity

Shawn Bowers
B.S., University of Oregon, 1998
M.S., OGI School of Science and Engineering at OHSU, 2003

A dissertation presented to the faculty of the
OGI School of Science and Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

January 2004

The dissertation "The Uni-Level Description: A Uniform Framework for Managing Structural Heterogeneity" by Shawn Bowers has been examined and approved by the following Examination Committee:

Lois Delcambre
Professor, OGI School of
Science and Engineering
Thesis Research Adviser

David Maier
Professor, OGI School of
Science and Engineering

Mark Jones
Associate Professor, OGI School of
Science and Engineering

Leonidas Fegaras
Associate Professor, University of
Texas at Arlington

Dedication

To Evening and Elizabeth

Acknowledgements

I wish to first thank Lois Delcambre, my advisor, whose help and guidance was invaluable to me throughout my five years of study at the Oregon Graduate Institute (OGI). Without her support, the work in this dissertation would not have been possible. Lois encouraged me to pursue a Ph.D. and introduced me to the subject of databases. She has helped me gain confidence as a researcher and to more effectively communicate my ideas. She also had the difficult task of keeping me on track. It has been a pleasure for me to have the opportunity to work with Lois. And I will always appreciate her patience and the time she has spent as a mentor to me.

Dave Maier also deserves special thanks. He participated in various discussions with Lois and I on issues surrounding meta-data-models and data models. Dave also carefully reviewed many of our papers, always providing detailed and insightful comments. I learned many things from Dave not only in his courses on Scholarship Skills, Object Data Management, and Principles of Database Systems (which Lois and Dave taught together), but also by working with him on papers and proposals, watching him present research, and seeing him discuss and debate ideas in research meetings. It is difficult to articulate the many ways that Lois and Dave have contributed to my personal and professional growth while I attended OGI. I will always be grateful for their help and advice.

My thesis committee also deserves many thanks, which along with Lois and Dave, included Mark Jones and Leo Fegaras. Both Mark and Leo provided valuable, detailed comments and feedback on this dissertation. I also appreciate each members flexibility in terms of scheduling my defense and in receiving drafts of the dissertation while I was in San Diego. In addition, I would like to thank Juliana Freire, Peter Heeman, and Renée Miller for participating in my thesis proposal.

There are many students at OGI, and particularly in the database group, that deserve thanks. I collaborated most closely with Mathew Weaver and Sudarshan Murthy, researching various aspects of superimposed information. Not only did I enjoy working with them, but I was exposed to many aspects of their research, which strengthened my knowledge and understanding in the area. I would also like to thank Vassilis Papadimos and William Howe for being genuinely interested in discussing new research ideas and for engaging in lively debates. Teaching the database-system course with Peter Tucker enriched my education at OGI, and our technical discussions concerning all aspects of database management helped me greatly. I also want to thank the students and faculty that participated in the database reading group for broadening my knowledge of research issues in data management and computer science in general.

Most importantly, I would like to thank my wife Evening who is my best friend and who always believed I could finish. And to my daughter Elizabeth, who could not wait until I was "done working." The rest of my family also deserves thanks: my grandfather for gracefully supporting my education and showing endless faith that I can accomplish whatever I set out to do, my grandmother for her support and understanding, and special thanks to my parents for their generous help and encouragement.

During the course of this work, I was supported in part through research grants provided by the National Science Foundation (grant numbers EIA-9983518 and IIS-9817492) and the Defense Advanced Research Projects Agency (contracts F30602-96-1-0302 and N66001-00-C-8032). Finally, I would like to thank Bertram Ludäscher, who graciously helped make it possible for me to finish writing this dissertation and to schedule my thesis defense while I was working for him at the San Diego Supercomputer Center.

Contents

List of Tables

List of Figures

Abstract

The Uni-Level Description: A Uniform Framework for Managing Structural Heterogeneity

Shawn Bowers

Supervising Professor: Lois Delcambre

Information management systems (such as database and knowledge-based systems) are based on data models, which provide basic structures for organizing and storing data. A number of data models are in use, and each provides a slightly different set of structures. For instance, information is represented as tables in the relational data model, as ordered trees in semi-structured data models (such as XML), and as directed graphs in semantic networks (such as RDF). With several distinct data models available, developers can select the most convenient representation for their particular need. However, the use of multiple data models introduces structural heterogeneity, making it difficult to combine information and exploit generic tools (for example, for querying or browsing).

This dissertation describes a new framework called the Uni-Level Description (ULD) that can accurately store and accommodate information from a broad range of data models. The ULD consists of a meta-data-model to describe the basic data structures used by a data model, and a uniform representation to store information within a source, including the data-model structures, schema (if present), and instance data. The ULD extends existing meta-data-models by allowing uniform access to schema and data, by permitting data models with non-traditional schema arrangements, and by providing a language for representing data-model constraints.

The two primary motivations for our work are (1) to study the basic structural capabilities offered by data models and (2) to define languages for enabling interoperability among sources with structural heterogeneity. We use the ULD to describe a wide range of data models, including those that allow optional and partial schema. We present a query and transformation language (based on Datalog) for accessing and converting information among heterogeneous sources. We demonstrate the flexibility of the transformation language by defining a number of structural mappings. And finally, we use the ULD as the basis for generic navigation, where a single set of operators can be used to discover and browse information in structurally heterogeneous sources.

Chapter 1

Introduction

Information management systems provide a range of tools for working with data. For example, most systems provide tools to create, manipulate, and query data, and to maintain data consistency. Although not always obvious, the most fundamental tool an information system provides is its *data model*—a fixed set of data structures for storing, organizing, and processing information.

A data model defines a language for representing both conceptual information (in a *schema*) and instance data. A schema describes a domain of interest and typically prescribes an organization for data. Schema and data are represented in a data model using named data structures, called *constructs* [79]. The challenge when creating a data model is to select an appropriate mix of constructs that can both conveniently represent a wide range of information and enable tools for efficiently managing data.

In this dissertation, we study the structuring capabilities of data models. A large number of data models exist and are being used, and each provides a different set of constructs for representing information. Part of our motivation for examining data models is to enable interoperability—we envision a generic framework that can support tools for exploiting information, regardless of the data model used for representation. Here, we focus on the structural differences among data models and provide languages for expressing and resolving these differences.

This chapter introduces the problem of structural heterogeneity, which arises in part from the use of different data models, describes existing approaches for managing structural heterogeneity, introduces our approach (which we call the *Uni-Level Description*), and concludes with an outline of the rest of this dissertation.

Movie

<u>mid</u> : *integer*	title : *string*	runtime : *integer*	genre : *string*	company : *string*
1	‘The Usual Suspects’	106	‘Thriller’	‘Gramercy’
2	‘Meet the Parents’	108	‘Comedy’	‘Universal’

Cast

<u>mid</u> : *integer*	<u>character</u> : *string*	<u>actor</u> : *string*
1	‘Roger Verbal Kint’	‘Kevin Spacey’
2	‘Jack Byrnes’	‘Robert De Niro’

Review

<u>mid</u> : *integer*	rating : *float*	<u>source</u> : *string*
1	8.7	‘IMDB’
2	7.0	‘IMDB’

Figure 1.1: An example of a relational schema and instance.

1.1 Structural Heterogeneity

Figures 1.1, 1.2, and 1.3 show similar schema and instance data represented in the relational [2, 33, 71], XML [22], and RDF [47] data models. We describe these data models below to highlight the basic structural differences that occur when similar information is represented in disparate data models.

One of the most widely used data models is the relational model. The primary construct of the relational model is a table, which is similar to a mathematical relation. As shown in Figure 1.1, a relational schema consists of a set of table definitions, typically called *relations*, containing attribute names and corresponding scalar types, called *domains*. A table instantiates a relation, and consists of tuples, where each tuple has exactly one value for each attribute. Figure 1.1 shows three table definitions and their corresponding tables. A table definition can designate one or more attributes as a primary key, where every tuple in the table has a distinct collection of values for the attributes. Primary keys are usually underlined, as shown in Figure 1.1. A foreign key may also be defined between attributes in two relations, where a set of values in one table contains key values in the other table, forming a value-based relationship between the tables. (Foreign keys could be defined for Figure 1.1, for example, between the Cast and Movie, and Review and Movie tables.)

The Extensible Markup Language (XML) is a more recent data model that is used for

DTD Schema

```
<!ELEMENT moviedb (movie*)>
<!ELEMENT movie (title,studio,genre*,actor*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT studio (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT actor (#PCDATA)>
<!ATTLIST actor role CDATA #REQUIRED>
```

XML Document

```
<?xml version="1.0"?>
<moviedb>
  <movie>
    <title>The Usual Suspects</title>
    <studio>Gramercy</studio>
    <genre>Thriller</genre>
    <actor role="Supporting Actor">Spacey, Kevin</actor>
  </movie>
  <movie>
    <title>Meet the Parents</title>
    <studio>Universal</studio>
    <genre>Comedy</genre>
    <actor role="Leading Actor">De Niro, Robert</actor>
  </movie>
</moviedb>
```

Figure 1.2: An example of an XML schema and instance.

information interchange. In XML, a document is represented as a text file that defines a labeled, ordered tree. Schema languages for XML are based on tree grammars, which specify regular expressions over node labels. An example schema for XML (expressed as a document type definition (DTD) [22]) and an instance document are shown in Figure 1.2. Schemas for XML consist of element and attribute definitions. An element represents a node in a tree that can contain a set of attribute-value pairs and has an ordered collection of children nodes, where each child node is either an element or character-string data.

The Resource Description Framework (RDF) is the primary representation scheme for the Semantic Web [11]. The RDF data model is based on directed graphs, similar to semantic networks [53, 82], and is designed for annotating Web-content with semantic information. An example schema and instance are shown in Figure 1.3. Nodes in the graph are called *resources*, which are connected using *properties*. In RDF, statements about

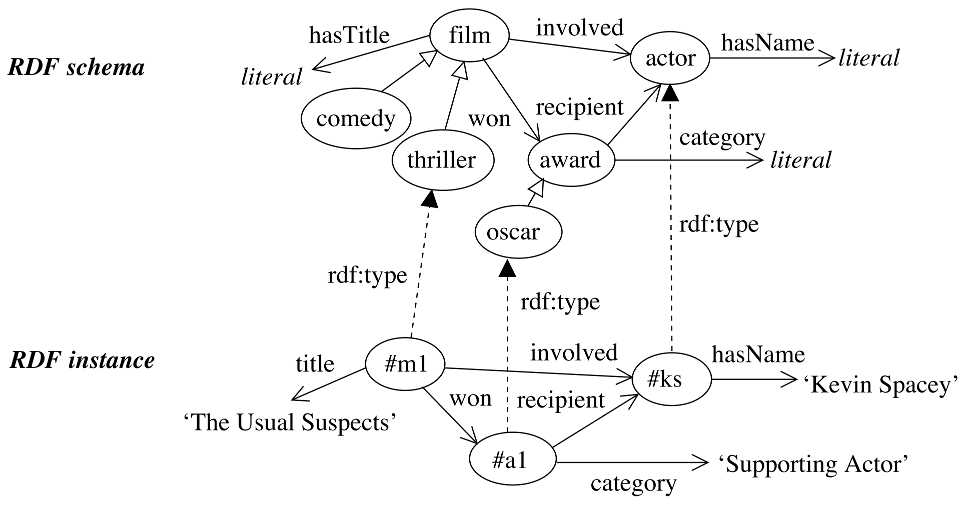

Figure 1.3: An example of an RDF schema and instance.

resources are made using *triples*, which relate a subject resource to an object resource through a property (sometimes called a *predicate* in a triple). RDF Schema (RDFS) [23] is a schema language for RDF that provides constructs for defining classes, subclasses, properties, and subproperties. As shown in Figure 1.3, the schema information is located at the top of the figure, where subclasses are drawn using a special relationship (for example, the "oscar" class is a subclass of "award"). Instances are located at the bottom of the figure, where relationships labeled with `rdf:type` connect instances to classes.

The relational model, XML, and RDF are structurally different data models. The relational model represents information using relations, XML using trees, and RDF using graphs.[1] One advantage of having several different data models is that users can select the right representation for their particular need, for example, based on the structural properties of the data they wish to represent. However, multiple representations introduce structural, model-based heterogeneity, making it difficult to combine information from different sources or to exploit information using generic tools, for example, for querying or browsing.

[1]Although not discussed in the previous examples, these data models also have fundamental differences in the way schemas restrict instance data, which we describe further throughout this dissertation.

Structural heterogeneity [40] can occur at various levels within a database. The presence of different data models introduces fundamental differences in how similar information is structured. Different schemas can also exhibit structural heterogeneity. Namely, two information sources may contain data from similar domains, but differ in how each schema organizes and classifies information. For example, the schemas for XML and RDF shown in Figures 1.2 and 1.3, respectively, have the following differences, even though they describe the same basic domain.

- Each schema includes information not found in the other. For example, the XML schema defines studio information (that is, the company that created the movie), whereas the RDF schema defines award information (that is, the awards won by the movie).

- The schemas represent conceptually identical information using different names and structural types. For example, the XML schema uses the label "movie," whereas the RDF schema uses "film." In addition, the RDF schema defines a film title as a direct property of a film and a film is allowed to have multiple titles. In the XML schema, a title is modeled as a sub-element of a movie and a movie has exactly one title. Note also that, in the RDF example, film and title are represented using different data-model constructs, whereas in the XML example, the same data-model construct is used to represent title and movie.

- The schemas contain *schematic* differences [57], where schema information in RDF is represented as data in XML. In particular, the RDF schema represents a genre as a class in the schema, whereas in XML a genre is represented as a data item (in this case, as plain text).

Finally, individual data values can also exhibit structural heterogeneity. For example, in the XML instance of Figure 1.2, the names of actors are stored last-name first, whereas in the RDF instance, the names of actors are stored first-name first. This example is a simple case, and in general, differences in data values can be complex.

1.2 Managing Structural Heterogeneity

Most approaches for managing structural heterogeneity focus on resolving differences in schema as opposed to differences in data models. These approaches, which we discuss here, resolve heterogeneity by providing support for either *schema transformation* or *schema integration*.

Schema transformation approaches [1, 12, 32, 35, 68, 69] provide techniques to convert valid instance data under one schema to valid instance data under another schema, where the source and target schemas are assumed to be heterogeneous. These approaches typically define a special-purpose, query-based transformation language to express schema mappings that, when executed, perform the desired instance-data conversion. In addition, a number of approaches attempt to determine schema mappings [70] automatically (or semi-automatically) by finding matching names and structures within the schemas.

Schema integration approaches [10, 24, 38, 40, 52, 62, 64, 75] focus on combining information sources with heterogeneous schemas into a single collection of data with a single schema (called a *global* schema). Many approaches [24, 52, 62, 75] construct a virtual integrated collection, where a query against the global schema is executed by a *mediator*, which rewrites the query into one or more local queries (using the local schemas), combines the results of each local query, and returns the integrated result. Schema integration requires schema transformation to resolve heterogeneity. In particular, schema mappings are defined as views (expressed as queries) between local and global schemas.

Approaches for schema transformation and integration typically assume all schemas and (possibly) data are represented in, or converted to the same, common data model. The common data model is often designed specifically for the approach.

In general, however, developers still use *ad hoc* solutions for resolving structural heterogeneity [12]. By *ad hoc*, we mean special-purpose programs, typically written in procedural languages, for resolving specific instances of structural heterogeneity. Even in schema transformation and integration approaches, special-purpose programs are used to map between data models, for example, to convert information in and out of a common data model. Writing programs to convert information between data models is often a time-consuming and error-prone task that leads to complex programs that are difficult to maintain and reuse. The following list highlights some of the reasons why converting information between data models, especially using *ad hoc* solutions, is difficult. We use the term *data-model mapping* to denote a mapping from information represented in one data model to information represented in another data model, which in general may also require schema mapping.

- *Data models contain different modeling constructs.* Although obvious, having disparate constructs means that mappings between data models are rarely one-to-one, and there are usually many different and equally "correct" ways to convert information between data models. For example, numerous approaches have been proposed for converting XML into the relational model [14, 77], where each conversion has benefit in certain situations.

- *There may be different versions of the same data model.* For example, many versions of the E-R data model have been defined [27, 28], each with slightly different features. In particular, some E-R models permit only binary relationships, whereas others allow n-ary relationships; and some E-R data models permit both entities and relationships to have attributes, whereas others do not allow attributes on relationships.

- *Desired mappings between data models may be partial.* In particular, some constructs or schema types may not be applicable to the target information source. In general, partial mappings result in "lossy" transformations, where the original database cannot be rebuilt from the converted information.

- *Mappings between data models may require encoding conventions.* Because data-model constructs differ, *conventions* are often used to represent the desired information within a target data model, when the target data model does not contain the necessary constructs. For example, conventions are heavily used in most XML-to-relational mappings, because the relational model does not have explicit constructs for representing trees. The use of conventions introduces implicit assumptions and dependencies between source and target databases.

- *Not all data models require schema, and some allow multiple levels of schema.* In general, data models that allow non-traditional schema arrangements are problematic for schema transformation and integration approaches, because not all data of interest may have corresponding schema information, and schemas may be further organized by meta-schema structures [43, 59]. By non-traditional, we mean schema arrangements not typically allowed in traditional data models, such as permitting partial, optional, and multiple levels of schema. Similarly, schematic differences and differences in how data is represented also generally require complex transformations [3, 57].

- *Data models have inherent constraints.* Resolving structural heterogeneity is complicated by data-model constraints. For example, *ad hoc* solutions must be aware of constraints to avoid incorrect conversions. Data-model constraints can range from simple, for example, relational tables have a corresponding table definition, to complex, for example, the restriction placed on tables under primary and foreign keys.

1.3 Our Approach

In this dissertation, we describe a high-level, declarative language for expressing and manipulating data models, called the *Uni-Level Description* (ULD) [19, 20]. With the ULD, it is possible to describe the constructs and constraints of a wide range of data models. In addition, the ULD provides uniform access to data-model constructs, schema, and instance-data, which enables flexible transformation rules for resolving structural heterogeneity and provides a means to implement generic operations over data models.

The ULD is motivated by a simple observation: most data models use only a small set of basic data structures, such as *atomic*, *tuple*, and *collection* types, composed in various ways, to store information. Our approach is to describe the constructs of a data model using these basic structures and then instantiate them to describe the schema and instance information that is present in a data source. The goals of the ULD are as follows.

- Permit "faithful" representations of a wide variety of data models. For example, the relational model should be described as having relations, tables, tuples, and so on, without requiring these constructs to be converted to the constructs of another, common data model.

- Allow a single and complete representation of an information source, including its data, schema(s) if present, and data model.

- Provide uniform access to information within a source. In particular, the ULD should be a uniform representation, much like RDF, so that query or navigation operators can access the data model and schema information as easily as data. More than that, the data model, schema, and data information should be accessible at the same time, for example, in the same transformation rule.

- Accommodate information with flexibility regarding the use of schemas. We want to represent information where the schema is required (as in the relational model), where schema can be missing (as in XML), and where there can be multiple levels of schema (as in RDF).

- Support data-model constraints. The ULD should provide an expressive enough language to capture inherent data-model constraints.

A key characteristic of the ULD is its use of explicit and distinct *instance-of* relationships for representing information. These relationships enable uniform access to information while clearly distinguishing between data-model constructs, schema (when appropriate), and data. This approach differs from data models that overload the *instance-of* (or *type*) relationship to provide uniform representation (such as RDF, Telos [43, 59], or other semantic-network-based data models). In addition, having distinct instance-of relationships allows a data-model configurer to place explicit constraints on schema-instance conformance, permitting accurate descriptions of a wide range of data models.

We envision an environment where arbitrary data sources are easily described using the ULD, enabling the use of generic, ULD-based tools that can work over arbitrary information sources, as shown in Figure 1.4. We describe two such tools that exploit the ULD

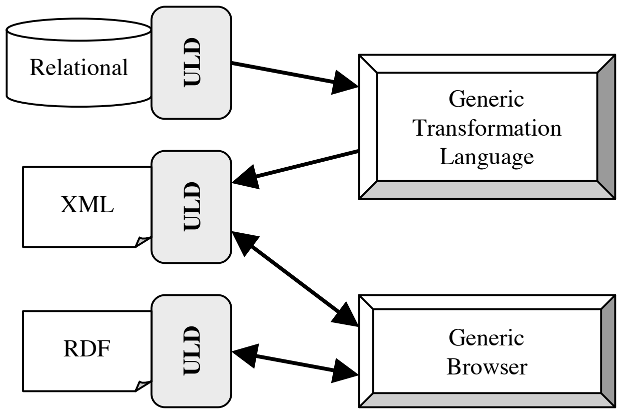

Figure 1.4: The ULD-enabled environment where generic tools can access information stored in various data models.

in this way. The first is a transformation language that allows powerful transformation rules expressed in Datalog (or Prolog) for defining and executing mappings to convert information from one source to another in the presence of different schemas or data models. The other is a browser, supported by a navigational API against the ULD, that permits both naïve users (for example, users who do not know, or need to know, what an XML element or attribute is) and sophisticated users (for example, agents or crawlers) to access information sources in a generic manner.

1.4 Outline

The rest of this dissertation is organized as follows. In Chapter 2 we describe the ULD and the associated languages for representing information sources, expressing queries (to retrieve information represented in the ULD), and specifying constraints. In Chapter 3 we demonstrate the ULD by giving complete descriptions for a wide range of data models. Chapter 3 also provides properties to classify data models in terms of the types of schema arrangements they allow. The classification is inspired by the ULD architecture and its flexibility in describing schema-instance relationships. In Chapter 4 we introduce the ULD transformation language, which is an application of the ULD architecture. We demonstrate the ability of the transformation language to express flexible mappings for resolving structural heterogeneity, including differences in data models and schemas. In Chapter 5 we describe a navigation interface for generically browsing information represented in disparate data models. The navigation interface is an example of a generic tool: the same operators are used to navigate information regardless of the data model used for representation. Finally, in Chapter 6 we conclude by discussing ULD implementation issues, related work, our contributions, and future work.

Chapter 2

The Uni-Level Description

In this chapter, we introduce the *Uni-Level Description* (ULD), which is both a *meta-data-model* (that is, it can describe data models), and a distinct representation scheme: it can directly represent both schema and instance information (expressed in terms of data model constructs). We also describe the ULD query and constraint languages, and give a formal definition of the underlying theory of the ULD.

This chapter is organized as follows. In Section 2.1, we describe existing meta-data-model architectures and contrast them with the architecture of the ULD. In Section 2.2, we present the ULD meta-data-model (that is, a specific set of basic structures for the ULD), which we use in the rest of this dissertation. In Section 2.3, we describe an implementation-independent syntax for representing information in the ULD and use it to describe the relational, XML, and RDF data models. In Section 2.4, we describe a declarative query language for the ULD based on Datalog. The ULD query language provides a convenient, high-level language for uniformly accessing data model, schema, and instance data. We also give a formal, logic-based interpretation of the ULD representation language. In Section 2.5, we describe a constraint language for the ULD, which extends the ULD query language of Section 2.4, and can be used to specify data-model constraints. Finally, in Section 2.6, we give an axiomatization for the ULD using the constraint language, which is enabled by the flexibility of the ULD architecture, and the ability of the query language to uniformly access constructs, schema, and data in the ULD.

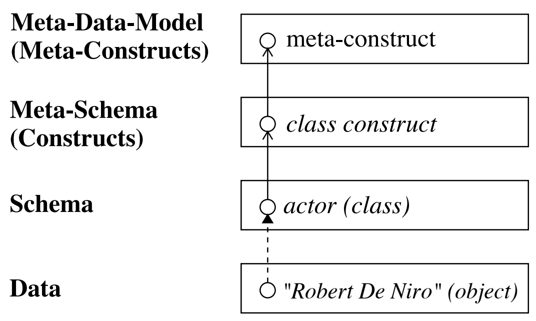

Figure 2.1: A standard meta-data-model architecture.

2.1 Architecture

2.1.1 Typical Meta-Data-Model Architectures

The standard architecture for representing information consists of three levels: data model, schema, and (instance) data. A *meta-data-model* adds a new level of abstraction to the top of this architecture to describe one or possibly more data models, as shown in Figure 2.1. (Note that the architecture of Figure 2.1 is more general than most existing architectures [5, 6, 8, 31, 32, 78].) As shown, the architecture consists of four levels. The top level represents the meta-data-model, the second level from the top represents the meta-schema (that is, the data model), the third level represents schema, and the bottom level represents instance data. A meta-schema defines the set of structural *constructs* provided by a data model for representing schema; and schemas are represented as instantiations of the meta-schema constructs. For example, the meta-schema for the relational data model would describe relations, attributes, primary and foreign keys, and so on.

In a *self-describing data model* [54], the meta-data-model and the data model are the same—that is, the same storage structures are used for schema and data. For example, a self-describing representation of the relational data model would encode a relational schema (that is, the set of relation names, their attributes, foreign and primary keys, and domains) using a fixed set of relations, called the *catalog*. The catalog can then be queried, for example, to find the attributes of a relation.

In contrast to a self-describing data model, a meta-data-model can be used to enable interoperability among heterogeneous sources that use disparate data models (as opposed to just the relational model, for example). In this case, each heterogeneous source is described using the same meta-data-model, which provides a common representation for accessing schema (via the meta-schema structures).

A number of approaches use E-R (entity and relationship) structures to define meta-schemas and their corresponding schemas [5, 6, 8, 31, 61, 78]. In this case, meta-schema constructs are defined as patterns, or compositions of entity and relationship types. These types are used to define entities and relationships representing corresponding schema items. Using this approach, a meta-schema for an object-oriented data model would contain a class construct represented as an entity type (called *class*) with a name attribute and relationships to other constructs (for example, that represent class attributes). A particular object-oriented schema is then represented as a set of entities and relationships that instantiate these meta-schema types. We note that, in most meta-data-model architectures (YAT [32] being an exception), data (the bottom level of Figure 2.1) is stored in the database system, and is not represented in the meta-data-model architecture explicitly.

An important assumption made by existing meta-data-model approaches is that data (the bottom level of Figure 2.1) must adhere to the schema (the third level of Figure 2.1). That is, all data is considered to be an instance of existing schema definitions. This assumption stems from traditional database systems, which generally require *complete schema*. That is, in a traditional database system, all data must satisfy all of the constraints imposed by a schema. However, a number of data models also permit *optional* and *partial* schema definitions, in which some or all of the data does not conform to a defined schema and, in the case of a partial schema, data items that conform to a schema can be mixed with data items that do not have associated schema definitions. Optional- and partial-schema data models cannot be represented with the assumptions underlying the architecture of Figure 2.1. Namely, there is no way to go from meta-schema directly to data without first going through schema—the meta-schema only defines structures for representing schema.

2.1.2 The ULD Architecture

In contrast to Figure 2.1, the ULD uses the architecture[1] shown in Figure 2.2. The ULD meta-data-model (shown as the top level in Figure 2.2) consists of *construct types* (that is, meta-constructs) that denote structural primitives. The middle level uses the structural primitives to define both data and schema *constructs*, possibly with *conformance relationships* between them. For example, in Figure 2.2, the *class* and *object* constructs are related through a conformance link (labeled *conf*) and their corresponding *construct instances* in the bottom layer (that is, *actor* and the object with the name 'Robert De Niro') are related through a data instance-of link (labeled *d-inst*). It is important to note that both schema and data constructs of a data model are explicitly represented in the ULD. In existing meta-data-model architectures, only schema constructs (as shown in Figure 2.1) are defined in the meta-schema—data constructs are assumed to implicitly follow from the definition of the meta-schema. The bottom level of Figure 2.2 represents schema and data as instances of constructs, connected by *d-inst* links when appropriate. The relationship between the ULD architecture and the architecture of Figure 2.1 is shown in Figure 2.3. In particular, the ULD architecture subsumes the typical meta-data-model approaches of Figure 2.1.

The ULD distinguishes three kinds of instance-of relationships. Constructs introduced in the middle layer are necessarily instances of construct types in the meta-data-model, represented with *ct-inst* instance-of links. Similarly, every item introduced in the bottom layer is necessarily an instance of a construct in the middle layer, represented with *c-inst* instance-of links. Finally, an item in the bottom layer can be an instance of another item in the bottom layer, represented with *d-inst* instance-of links, as allowed by *conformance* relationships specified in the middle layer. Note that we distinguish the term *instance* from the term *instance-of*. As an example, the bottom layer of the ULD architecture

[1]The ULD is a flat representation in that all information stored in the ULD is uniformly accessible as a single level. For example, a single query or transformation rule can access data-model constructs, schema, and instance data. The name *Uni-Level Description* was chosen to reflect this property of the ULD, that is, of being a flat representation. However, as shown in Figure 2.2, information stored in the ULD is *logically* divided into three layers using explicit instance-of relationships (which we describe in this chapter).

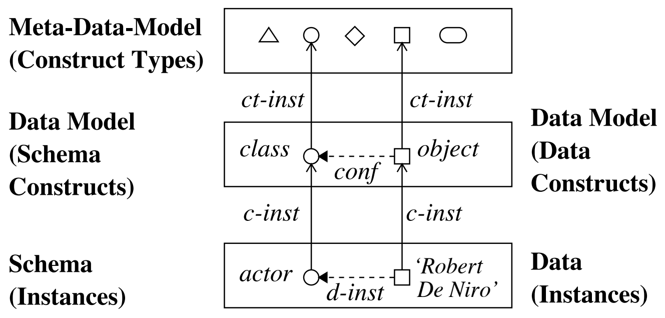

Figure 2.2: The ULD meta-data-model architecture.

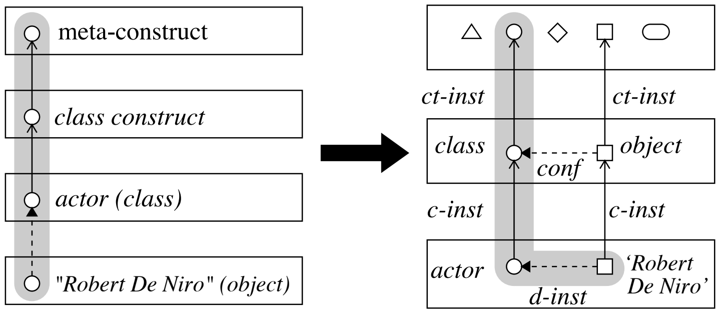

Figure 2.3: The standard meta-data-model architecture (left) captured in the ULD architecture (right).

consists of *construct instances* (that is, data and schema items), and items within the architecture are associated using *instance-of* relationships (that is, *ct-inst*, *c-inst*, and *d-inst*). A construct instance may be a *d-inst* of another construct instance (all in the bottom layer), and must be a *c-inst* of at least one construct (in the middle layer).

The ULD is able to represent a wide range of data models using the flexibility offered by the *conf* and *d-inst* relationships. For example, in XML, elements are not required to conform to element types. In this case, the XML element would not have a *d-inst* link to an XML element type (because there are no element types explicitly defined that the element conforms to). Figure 2.4 shows the case where data is represented in the ULD without corresponding schema.

The ULD represents an information source as a *configuration*, which contains the

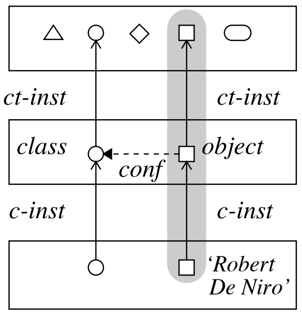

Figure 2.4: An example of data without corresponding schema represented in the ULD.

constructs of a data model, the construct instances (both schema and data) of a source, and the associated conformance and instance-of relationships. Thus, a configuration can be viewed as an instantiation of Figure 2.2. Each configuration uses a finite set of identifiers to denote construct types, constructs, and construct instances as well as a finite set of *ct-inst*, *c-inst*, *conf*, and *d-inst* relationships. Together, the use of identifiers with explicit instance-of relationships allows all three levels of information in Figure 2.2 to be stored in a single configuration—enabling direct and uniform access to all data-model, schema, and data information in a source. In a ULD configuration, all construct and construct-instance identifiers have exactly one associated value, where a value instantiates a meta-data-model structure.

2.2 The ULD Meta-Data-Model

This section briefly describes the choice of structures we use as the construct types of the ULD meta-data-model. We describe these structures in more detail in the following section.

The ULD meta-data-model contains primitive structures for tuples, that is, sets of name-value pairs; set, list and bag collections; atomics, for scalar values such as strings and integers; and unions, for representing non-structural, generalization relationships among constructs. The construct-type identifiers for these structures are denoted *struct-ct*, *set-ct*, *list-ct*, *bag-ct*, *atomic-ct*, and *union-ct*, respectively.

Scalar values such as strings and integers are treated as subsets of the construct-instance identifiers in a configuration. This choice of representation leads to more concise configurations, however, primitive values could also be represented as distinct values, disjoint from the set of identifiers.

Tuple and collection structures can be nested, but only through identifiers. For instance, set values always contain a unique collection of identifiers, as opposed to a unique collection of values. Thus, to construct a nested set value, we would create an identifier i_1 whose value is a set containing an identifier i_2, where i_2's value is a set.

The structures of the ULD meta-data-model are similar to those found in E-R [27] and object-oriented database (OODB) [48] data models. Unlike the E-R data model, however, the ULD does not treat relationships as first class; a relationship in the ULD is specified within a tuple structure, which is similar to some OODB data models. The ULD also has explicit structures for representing collections, which have identifiers and can stand alone or be nested within tuples and other collections. Thus, collections in the ULD serve to represent extents, set-valued relationships, and collection-valued attributes in OODB data models. Finally, the ULD uses union types, and does not provide *isa* relationships between structures, which are found in OODB and some E-R data models. We note that other meta-data-models, for example, with different structures or allowable compositions of structure, are possible without requiring changes to the ULD architecture of Figure 2.2. However, the particular meta-data-model we have chosen is rich enough to express a wide variety of existing data models (as we show in Chapter 3) without requiring complex data structures and nested values, which require a more complex representation and query language.

2.3 The ULD Representation Language

In this section, we describe an implementation-independent language for representing configurations in the ULD. By implementation-independent, we mean a language that can be used regardless of the underlying storage of a configuration, for example, a relational database or a set of facts in a deductive database. Thus, the ULD representation language

$$\begin{array}{lll}
E & ::= & C \mid D \\
 & & \\
C & ::= & \texttt{construct}\ c\ \texttt{=}\ (\ CS \mid CC \mid CA \mid CU\ ) \\
CS & ::= & \texttt{\{}a_1\texttt{->}c_1\texttt{, ..., }a_n\texttt{->}c_n\texttt{\}}\ [\ F\ ] \\
CC & ::= & (\ \texttt{set} \mid \texttt{bag} \mid \texttt{list}\ )\ \texttt{of}\ c_1\ [\ F\ ] \\
CA & ::= & \texttt{atomic}\ [\ F\ ] \\
CU & ::= & c_1\ \texttt{|}\ c_2\ \texttt{|}\ \texttt{...}\ \texttt{|}\ c_n \\
F & ::= & \texttt{conf(domain=}x\texttt{, range=}y\texttt{):}c' \\
 & & \\
D & ::= & \texttt{data}\ d\ \texttt{=}\ (\ DS \mid DC \mid DA\ )\ [\ I\ ] \\
DS & ::= & c\ \texttt{\{}a_1\texttt{:}d_1\texttt{, ..., }a_n\texttt{:}d_n\texttt{\}} \\
DC & ::= & c\ \texttt{[}d_1\texttt{, ..., }d_n\texttt{]} \\
DA & ::= & c \\
I & ::= & \texttt{d-inst:}d_1\texttt{, }d_2\texttt{, ..., }d_n
\end{array}$$

Figure 2.5: The syntax for ULD construct and instance expressions.

can be viewed as a high-level syntax for defining ULD configurations. To illustrate, the following examples (Examples 1, 2, and 3 below) use the ULD representation language to describe simplified versions of the relational, XML, and RDF data model constructs, respectively.[2] (Chapter 3 defines these and other data models in more detail.) Note that there are potentially many ways to describe a data model in the ULD, and these examples show only one choice of representation.

As illustrated in Examples 1, 2, and 3, a construct definition can take one of the following forms. Expressions surrounded by [ ] square brackets are optional and expressions of the form $(a|b|c)$ denote a choice between expressions a, b, or c. The syntax for expressing constructs and data instances in the ULD, each represented as a ULD expression E, is given in Figure 2.5. Note that the symbols `[ ]` and `|` are tokens in the language, whereas [ ] and | are not.

- `construct` c `=` $\texttt{\{}a_1\texttt{->}c_1\texttt{,}a_2\texttt{->}c_2\texttt{,...,}a_n\texttt{->}c_n\texttt{\}}$ [$\texttt{conf(domain=}x\texttt{,range=}y\texttt{):}c'$]

  This expression defines a tuple-construct c as a *ct-inst* of construct type *struct-ct*, where a_1 to a_n are distinct strings, $n \geq 1$, and c_1 to c_n and c' are construct identifiers. Each expression $a_i\texttt{->}c_i$ is called a *component* of the construct c where a_i is called the

[2]We use `uldValue` and `uldValuetype` as special constructs to denote scalar values and value types, which we describe later.

component *selector*. If the *conf* expression is present, construct instances of c may conform (that is, be connected by a *d-inst* relationship) to construct instances of c' according to the domain and range constraints on the expression. If the conformance expression is not present, conformance is not permitted for the construct, that is, there cannot be a *d-inst* relationship for the construct's instances. The cardinality constraints on conformance, shown as x and y above, restrict the participation of associated instances in *d-inst* relationships, for both the domain and range of the relationship, to either: exactly one, denoted as `1`; zero or one, denoted as `?`; zero or more, denoted as `*`; or one or more, denoted as `+`.

- `construct` c `=` (`set` | `bag` | `list`) `of` c_1 [`conf(domain=`x`,range=`y`):`c']

  This expression defines a collection-construct c as a *ct-inst* of either *set-ct*, *bag-ct*, or *list-ct*. The definition restricts c's construct instances to collections whose members must be instances of the construct c_1. In addition, for instances of a set construct c, each member must have a distinct identifier.[3] This construct can also contain a conformance definition.

- `construct` c `= atomic` [`conf(domain=`x`,range=`y`):`c']

  This expression defines an atomic type c as a *ct-inst* of construct type *atomic-ct*. This construct can also contain a conformance definition.

- `construct` c `=` c_1 `|` c_2 `| ... |` c_n

  This expression defines a union-construct c as a *ct-inst* of construct type *union-ct*, where c_1 to c_n are distinct construct identifiers for $n \geq 2$ such that, for all c_i, for $1 \leq i \leq n$, $c \neq c_i$. We also require acyclic union definitions, that is, no two constructs can be directly (or indirectly) unions of each other. Each instance of c_1 to c_n is considered an instance of c, however, we do not allow instances of only c directly. A union construct provides a simple mechanism to group heterogeneous structures (for example, atomics and structs) into a single, union type, as opposed

[3]We could similarly define a construct type, for example, *val-set*, that would require unique values, using deep equality [48], as opposed to identifiers, using shallow equality [48]. For simplicity, we only give identifier-based sets, but provide a language for expressing other constraints such as value restrictions on sets.

to *isa* relationships, which offer inheritance semantics, and only group like structures (for example, classes).

For the relational data model shown in Example 1, tables and relation types are one-to-one such that each table conforms to exactly one relation type and vice versa. Similarly, each tuple in a table must conform, as shown by the range restriction of `1`, to the table's associated relation type. We assume each relation type can have at most one primary key, for simplicity. Note that in the example below, we do not define the constraints implied by a key. In Chapter 3, we discuss the use of data-model constraints in more detail. (Example schema and instance data for the relational data model are presented later, in Example 4.)

Example 1 (*The relational data model*)

```
% schema constructs
construct relation    = {hasName->uldString, hasAtts->attList}
construct attList     = list of attribute
construct attribute   = {hasName->uldString, hasDomain->uldValuetype}
construct pKey        = {forRel->relation, keyAtts->pKeyAttList}
construct fKey        = {forRel->relation, toRel->relation, keyAtts->fKeyAttList}
construct pKeyAttList = list of attribute
construct fKeyAttList = list of attribute
% data constructs
construct table       = bag of tuple conf(domain=1,range=1):relation
construct tuple       = list of uldValue conf(domain=*,range=1):relation
```

The XML data model [22] shown in Example 2 includes constructs for element types, attribute types, elements, attributes, content models, and content, where element types contain attribute types and content specifications, elements can optionally conform to element types, and attributes can optionally conform to attribute types. We simplify content models to sets of element types for which a conforming element must have at least one subelement for each corresponding type.

Example 2 (*The XML with DTD data model*)

```
% schema constructs
construct pcdata     = atomic
construct cdata      = atomic
construct elemType   = {hasName->uldString, hasAtts->attDefList,
                       hasModel->contentDef}
construct attDefList = set of attDef
construct attDef     = {hasName->uldString}
construct contentDef = set of elemType
% data constructs
construct element    = {hasTag->uldString, hasAtts->attSet, hasChildren->content}
                       conf(domain=*,range=?):elemType
construct attSet     = set of attribute
construct attribute  = {hasName->uldString, hasVal->cdata}
                       conf(domain=*,range=?):attDef
construct content    = list of node
construct node       = element | pcdata
```

Finally, Example 3 shows the RDF data model with RDF Schema (RDFS) [23, 47] expressed in the ULD. The description includes constructs for classes, properties, resources, and triples. A triple in RDF contains a subject, predicate, and object, where a predicate can be an arbitrary resource, including a defined property. In RDFS, `rdf:type`, `rdfs:subClassOf`, and `rdfs:subPropertyOf` are considered special RDF properties for denoting instance and specialization relationships. However, we model these properties using conformance (for `rdf:type`) and the constructs `subClass` and `subProp`. For example, a subclass relationship is represented by instantiating (using *c-inst*) a `subclass` construct as opposed to using the special `rdfs:subClassOf` RDF property. This approach allows RDF properties and structural relationships to be decoupled. That is, in the ULD representation, RDF properties only model standard relationships instead of being overloaded to represent both standard relationships and type, subclass, and subproperty information. This approach does not limit the expressibility of RDF: partial, optional, and multiple levels of schema are still possible.

Example 3 (*The RDF(S) data model*)

```
% schema constructs
construct resource  = rdfType | simpleRes
construct rdfType   = class | prop
construct simpleRes = {hasURI->uldURI} conf(domain=*,range=*):class
construct class     = {hasURI->uldURI, hasLabel->uldString}
                      conf(domain=*,range=*):class
construct prop      = {hasURI->uldURI, hasLabel->uldString, hasDomain->class,
                      hasRange->rangeVal} conf(domain=*,range=*):rdfType
construct rangeVal  = class | uldValueType
construct subClass  = {hasSub->class, hasSuper->class}
construct subProp   = {hasSub->prop, hasSuper->prop}

% data constructs
construct triple    = {hasPred->resource, hasSubj->resource, hasObj->objVal}
construct objVal    = resource | literal
construct literal   = atomic
```

Every ULD configuration can be initialized with default constructs that represent typical primitive value types, such as *String*, *Boolean*, *Integer*, *URI*, etc., as instances of *atomic-ct*. In addition, `uldValuetype` and `uldValue`, as shown in Figure 2.6, are special constructs that work together to provide a mechanism for data models to permit user-defined primitive types (for example, to support relational domains). (Note that in Figure 2.6, we shorten `uldValue` to `value` and `uldValuetype` to `valuetype`.) The `uldValue` construct is defined as the union (that is, a union construct) of all defined *atomic-ct* constructs. Thus, when a new *atomic-ct* construct is created, it is added to `uldValue`'s definition.[4] The `uldValuetype` construct has an instance with the same name, represented as a string value, as each construct of `uldValue`. For example, to add a new primitive type for date values, we would create a new construct with the identifier *date*, assign the construct to `uldValue` (recall `uldValue` is a union construct), and create a new `uldValuetype` instance `'date'` (see Figure 2.6).

Sample schema and data (that is, the bottom level of Figure 2.2) for the relational, XML, and RDF data models are shown in Examples 4, 5, and 6, respectively. Example 4

[4]We assume that the construct is added by the system managing the ULD configuration, and not by the user.

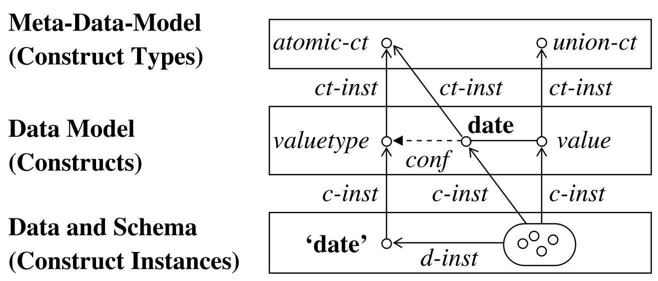

Figure 2.6: The default `uldValue` and `uldValuetype` constructs for representing scalar types.

gives a subset of the schema and data from Figure 1.1, Example 5 gives a subset of the schema and data from Figure 1.2, and Example 6 gives a subset of the schema and data from Figure 1.3. As shown, expressions for defining construct instances take the form: i `=` c v `d-inst:`i_1`,`i_2`,...,`i_l, where c is a construct, v is a valid value (see below) for construct c, and i_1 to i_l are construct-instance identifiers for $l \geq 0$. The expression defines i as a *c-inst* of c with value v. If c participates in a union construct c', i is considered a *c-inst* of both c and c'. Further, i is a *d-inst* of each construct instance in i_1 to i_l, each of which must be a *c-inst* of the construct that c conforms to. The `d-inst` keyword is omitted if i does not participate in a *d-inst* relationship with another construct instance (that is, $l = 0$).

As described above, a syntactically well-formed construct instance is allowed in a configuration only if it satisfies the constraints of its associated construct. We end this section by giving an informal definition of when a value is valid (with respect to its associated construct).

Given the construct c, we write $c \in \textit{ct-extent-of}(ct)$ if and only if c is defined as a *ct-inst* of construct type ct. Let $\textit{ct-extent-of}(ct)$ be the set of all instances in a configuration of construct type ct. Similarly, let $\textit{c-extent-of}(c)$ be the set of all construct instances in a configuration that are a *c-inst* of construct c, and $\textit{d-extent-of}(d)$ be the set of all construct instances in a configuration that are a *d-inst* of d. The expressions $i \in \textit{c-extent-of}(c)$ and $i \in \textit{d-extent-of}(i')$ are true if and only if i is defined as a *c-inst* of construct c and a *d-inst* of i'. The relation $\mathit{conf}(c, c', x, y)$ is true if and only if a conformance relationship is defined from c to c' (that is, instances of construct c can be a *d-inst* of c') with a domain constraint of x and a range constraint of y. The following constraints are imposed by constructs on their instances and must hold for a value to be valid. We note that each construct instance can only be defined once in a configuration, and must have exactly one associated, non-union construct, that is, it must be a *c-inst* of exactly one construct that is not a *ct-inst* of *union-ct*.

- **Structural Constraints**. Let i be a construct instance such that $i \in$ *c-extent-of*(c). If c is a tuple construct (that is, $c \in$ *ct-extent-of*(*struct-ct*)), the value of i must contain the same number of components as c, each component selector of i must be a distinct selector of c, and each component value of i must be an instance of the associated component construct value for c. (Note that we require all components to have a non-null value.) If c is a collection construct (that is, $c \in$ *ct-extent-of*(*set-ct*) $\cup$ *ct-extent-of*(*bag-ct*) $\cup$ *ct-extent-of*(*list-ct*)), the value of i must be a collection whose members are instances of the construct that c is a collection of, and if c is a set, i's elements must have distinct identifiers. If c is an atomic construct (that is, $c \in$ *ct-extent-of*(*atomic-ct*)), we assume that the value of i satisfies c. For example, if c is a string, i is a well-formed string; if c is an integer, i is a well-formed integer; and so on. We note that, if c is a union construct (that is, $c \in$ *ct-extent-of*(*union-ct*)), c itself does not impose any additional restrictions on i's value.

- **Conformance Constraints**. If i is an instance of construct c (that is, $i \in$ *c-extent-of*(c)), i' is an instance of construct c' (that is, $i' \in$ *c-extent-of*(c')), and c can conform to c' (that is, $\mathit{conf}(c, c', x, y)$ is true for some x, y), i is allowed to be a *d-inst* of i' (that is, we allow $i \in$ *d-extent-of*(i')). Further, domain and range cardinality constraints on conformance restrict the allowable *d-inst* relationships as follows. If the domain constraint on the conformance relationship is `+` (that is, $x =$`+`), each i' in the configuration must participate in at least one *d-inst* relationship, where there is at least one i for each i' such that $i \in$ *d-extent-of*(i'). If the domain constraint on the conformance relationship is `1`, each i' in the configuration must have exactly one

i such that $i \in$ *d-extent-of*(i'). And, if the domain constraint on the conformance relationship is `?`, each i' in the configuration has at most one i such that $i \in$ *d-extent-of*(i'). Similarly, if the range constraint on the conformance relationship is `+` (that is, $y =$`+`), every i in the configuration must be the *d-inst* of at least one i', that is, there must be at least one i' such that $i \in$ *d-extent-of*(i'). If the range constraint on the conformance relationship is `1`, each i in the configuration must be a *d-inst* of exactly one i'. Finally, if the range constraint on the conformance relationship is `?`, each i can be the *d-inst* of at most one i'.

Example 4 (*Sample relational data and schema*)

```
% schema
movie = relation {hasName:'movie', hasAtts:al1} al1 = attList
[a1,a2,a3,a4,a5] a1 = attribute {hasName:'mid', domain:'uldInteger'}
a2 = attribute {hasName:'title', domain:'uldString'} a3 = attribute
{hasName:'runtime', domain:'uldInteger'} a4 = attribute
{hasName:'genre', domain:'uldString'} a5 = attribute
{hasName:'company', domain:'uldString'} key1 = pKey {forRel:movie,
keyAtts:pkal1} pkal1 = pKeyAttList [a1] cast = relation
{hasName:'cast', atts:al3} al3 = attlist [a6,a7,a8] a6 = attribute
{hasName:'mid', domain:'uldInteger'} a7 = attribute {hasName:'actor',
domain:'uldString'} a8 = attribute {hasName:'character',
domain:'uldString'}
% data
tbl1     = table [t1] d-inst:movie
t1       = tuple [1,'The Usual Suspects',106,'thriller','Gramercy']
           d-inst:movie
tbl2     = table [t2] d-inst:cast
t2       = tuple [1,'Roger Verbal Kint','Kevin Spacey'] d-inst:cast
```

Example 5 (*Sample XML with DTD data and schema*)

```
% schema
movie   = elemType {hasName:‘movie', hasAtts:nilad, hasModel:moviecm}
nilad   = attDefSet []
moviecm = contentDef [title, studio, genre, actor]
```

```
title    = elemType {hasName:‘title’, hasAtts:nilad, hasModel:nilcm}
nilcm    = contentDef []
genre    = elemType {hasName:‘genre’, hasAtts:nilad, hasModel:nilcm}
actor    = elemType {hasName:‘actor’, hasAtts:actorat, hasModel:nilcm}
actorat = attDefList [role]
role     = attdef {name:‘role’}
% data
e1       = element {hasTag:‘movie’, hasAtts:nilas, hasChildren:e1cnt}
           d-inst:movie
nilas    = attSet []
e1cnt    = content [e2,e3,e4]
e2       = element {hasTag:‘title’, hasAtts:nilas, hasChildren:e2cnt}
           d-inst:title
e2cnt    = content [‘The Usual Suspects’]
e3       = element {hasTag:‘genre’, hasAtts:nilas, hasChildren:e3cnt}
           d-inst:genre
e3cnt    = content [‘thriller’]
e4       = element {hasTag:‘actor’, hasAtts:e4as, hasChildren=e4cnt}
           d-inst:actor
e4as     = attSet [a1]
e4cnt    = content [‘Kevin Spacey’]
a1       = attribute {hasName:‘role’, hasVal:‘supporting’} d-inst:role
```

Example 6 (*Sample RDF(S) data and schema*)

```
% schema
film       = class {hasURI:‘#film’, hasLabel:‘film’}
title      = prop {hasURI:‘#title’, hasLabel:‘hasTitle’, hasDomain:film,
             hasRange:‘literal’}
thriller   = class {hasURI:‘#thriller’, hasLabel:‘thriller’}
filmthril = subclass {hasSub:thriller, hasSuper:film}
% data
f1 = simpleRes {hasURI:‘http://.../review.html’} d-inst:thriller,film
t1 = triple {hasPred:title, hasSubj:f1, hasObj:‘The Usual Suspects’}
```

2.4 The ULD Query Language

This section describes a query language for the ULD that exploits the capabilities of Datalog [2] (we describe Datalog in more detail below). A ULD query is expressed as a

Datalog program, that is, a set of restricted horn-clause rules, and is executed against a particular configuration. As an example, the following query finds all available class names within an RDF configuration. Note that upper-case terms denote variables and lower-case terms denote constants. Also note that the expression C.hasLabel:X, which is non-standard Datalog syntax, selects the hasLabel component-value X from the class-construct instance C. We describe additional extensions to Datalog in Section 4.2.2.

classname(X) ← c-inst(C, class), C.hasLabel:X.

The rule is read as "If C is an RDF class and the label of C is X, then X is a classname." As another example, the following query returns the property names of all classes in an RDF configuration. Note that this query, like the previous one, is expressed solely against the schema of the source, that is, we are only accessing classes and properties.

hasProp(X, Y) ← c-inst(C, class), c-inst(P, prop), P.hasDomain:C, C.hasLabel:X, P.hasLabel:Y.

After finding the available RDF classes and their corresponding properties, we can then use this information to find corresponding data instances. That is, a user could issue the previous query, see that the source contains title properties, and then construct the following query to return all title values in the configuration.

filmtitle(X) ← c-inst(P, prop), P.hasLabel:'title', c-inst(T, triple), T.hasPred:P, T.hasObj:X.

With the ULD, it is possible to query data (including accessing data without first accessing schema), schema (as shown in the previous queries), and data-model constructs. For example, the following query is expressed directly against data, and returns the URI of all RDF resources used as a property in at least one triple (note that the resource may or may not be associated with schema).

dataprop(X) ← c-inst(T, triple), T.hasPred:P, P.hasURI:X.

Once this query is executed, we may wish to find additional information about a particular resource. For example, the following query returns all values of the title resource used as the property of a triple. (Note that, in RDF, resources are typically represented with a leading '#' symbol.)

propval(X) ← c-inst(T, triple), T.hasPred:P, P.hasURI:'#title', T.hasObj:X.

The following queries are similar to the previous examples, but are expressed against an XML configuration. The first query finds the names of all available element types in the source, the second finds, for each element-type name, its corresponding attribute-definition names, the third finds the set of movie titles, the fourth finds all available attribute names as a data query, and the last query finds the set of attribute values for title attributes in the configuration. The membership operator $\in$ in the second query is used to access elements within a collection.

elemtype(X) ← c-inst(E, elemType), E.hasName:X.

atttype(X,Y) ← c-inst(E, elemType), E.hasName:X, E.hasAtts:AL, AT$\in$AL, AT.hasName:Y.

title(X) ← c-inst(AT, attDef), AT.hasName:'title', d-inst(A, AT), A.hasVal:X.

atts(X) ← c-inst(A, attribute), A.hasName:X.

titleattval(X) ← c-inst(A, attribute), A.hasName:'title', A.hasVal:X.

Finally, queries can be posed against data-model constructs directly, for example, to find the constructs used by an information source. The following query returns all constructs that serve as *struct-ct* schema constructs and their component selectors. Note that the expression SC.P->C is similar to the expression X.P:Y, but applies to constructs instead of construct instances.

schemastruct(SC, P) ← ct-inst(SC, struct-ct), conf(DC, SC, X, Y), SC.P->C.

Similarly, it is possible to access data values within a configuration, without knowledge of the data model. For example, the following query returns all primitive values of *struct-ct* constructs that serve as data for a corresponding schema item within a configuration.

dataval(V) $\leftarrow$ ct-inst(DC, struct-ct), conf(DC, SC, X, Y), DC.S->C,
ct-inst(C, atomic-ct), c-inst(D, DC), D.S:V.

The remainder of this section describes, in more detail, the use of Datalog for expressing queries over ULD configurations. We begin by giving a brief introduction to Datalog in Section 2.4.1, then describe an approach for interpreting a ULD configuration as a source for Datalog queries in Section 2.4.2. In Section 2.5 we give a method for defining configuration constraints, which are expressed as Datalog programs.

2.4.1 Datalog

Datalog is a logic-based language for querying relational databases. A Datalog query (or program) consists of a finite set of *Datalog rules*. Each Datalog rule takes the following form, for $n \geq 1$.

$$R_1(u_1) \leftarrow R_2(u_2), R_3(u_3), ..., R_n(u_n).$$

This rule is read as "$R_2(u_2)$ and $R_3(u_3)$ and ... and $R_n(u_n)$ implies $R_1(u_1)$." In a Datalog rule, each R_i represents a relation and each u_i represents a term of R_i. Often, R_i is referred to as a *predicate* applied to terms u_i, where the expression $R_i(u_i)$ is referred to as an *atomic formula* (or just *formula*) and u_i consists of a list of *terms* $u_i = u_{i1}, u_{i2}, ..., u_{im}$ for $m \geq 1$. We use the standard convention where predicates and constants begin with lower-case letters (or possibly special characters) and variables begin with upper-case letters. Datalog rules are *range restricted*, that is, each variable appearing in u_1 must appear in $u_2, ..., u_n$. The *head* of the rule occurs on the left side of the arrow and always contains a single formula $R_1(u_1)$. The *body* of the rule consists of the set of conjoined formulas $R_2(u_2), ..., R_n(u_n)$.

A Datalog program contains finite sets of *extensional* and *intensional* relations, whose instances (that is, sets of associated facts) are called the *extensional database* (EDB) and *intensional database* (IDB), respectively. An extensional relation is a relation that exists in the underlying database and can only occur in the body of a rule. Thus, an extensional formula $R_i(u_i)$ corresponds to the relation in the underlying database with the name R_i, and u_i represents a tuple (as a list of terms) with the same arity as the corresponding database relation. An intensional relation is a relation occurring in the head of some rule. The result of a executing a Datalog program is an IDB.

Equality and inequality predicates ($=$, $\neq$, $<$, $>$, and so on) are treated as special relations in Datalog. However, the introduction of equality further constrains the range-restriction requirement. Namely, in every equality expression containing two variables, at least one variable must occur as a variable in a relational atom $R_i(u_i)$ in the body of the rule. For an inequality, both variables must occur in a relational atom in the body of the rule. We note that equality and inequality relations are typically represented using infix notation, that is, $x = y$ instead of $=(x, y)$. Similar to equality and inequality relations, Datalog can also be extended to support functions [80]. The ULD query language uses functions and relations to access configurations, which we describe further in Section 2.4.2.

In the *minimal model semantics*, the answer to a Datalog program P over input I (where I represents the given instances, or tuples, of the extensional relations) is defined as the minimal model of P containing I. Alternatively, the *least fixpoint semantics* states that an answer to P is the least fixpoint of P, which is computed by repeatedly applying the rules of P on input I. Every Datalog program, as defined above, has a least fixpoint, which is also the minimal model.

As defined, only positive (that is, non-negated) formulas are permitted in a Datalog rule. Datalog$\neg$ extends Datalog by allowing negative formulas in the body of rules.[5] Similar to Datalog, every variable in the head of a *range-restricted* Datalog$\neg$ rule appears in some positive relation occurrence in the body. In addition, every variable in a negated formula of a Datalog$\neg$ rule must occur in a positive formula in the body of the rule. We note that not all Datalog$\neg$ programs have a unique least fixpoint.

A restricted form of Datalog$\neg$ considers only *stratified* programs, which place a syntactic restriction on rules. (The ULD query language assumes queries are expressed as stratified Datalog$\neg$ programs.) Namely, a stratified program can be partitioned into a finite number of sets (or layers) of rules $P^1, P^2, ..., P^n$ such that (1) for each IDB relation

[5]Note that Datalog$\neg$ uses the closed-world assumption, where facts that are not explicitly stated as true or that cannot be inferred are assumed to be false.

R, all rules in P defining R are in the same partition, (2) if $R(u) \leftarrow ...R'(v)...$ is a rule in P, R' is an IDB relation, R' is in partition P^i, and R is in partition P^j, then $i \leq j$, and (3) if $R(u) \leftarrow ...\neg R'(v)...$ is a rule in P, R' is an IDB relation, R' is in partition P^i, and R is in partition P^j, then $i < j$. Stratification gives a natural order for executing rules, namely, the fixpoint of partition P^1 is computed first, followed by the fixpoint of partition P^2, and so on, where partition P^n is computed last. A program that can be expressed using stratified Datalog¬ is guaranteed to have a unique least fixpoint. However, not all Datalog¬ programs can be stratified.

Finally, we note that it is possible to construct an unsatisfiable query in Datalog¬, that is, regardless of the EDB, the query always has an empty result. For example, the rule $p(\mathrm{X}) \leftarrow q(\mathrm{X}), \neg q(\mathrm{X})$ will never return an X that makes $p(\mathrm{X})$ true.

2.4.2 Querying ULD Configurations using Datalog

As mentioned in the beginning of this section, a ULD query is expressed using Datalog, and more precisely, using stratified Datalog¬. Thus, we treat a configuration $\mathcal{F}$ as a database instance consisting of a fixed set of extensional relations. To describe a configuration as an EDB, we represent a configuration $\mathcal{F}$ as a tuple $(\mathcal{I}_{\mathcal{CT}}, \mathcal{I}_{\mathcal{C}}, \mathcal{I}_{\mathcal{D}}, \mathcal{V}, val,$ *ct-extent-of*, *c-extent-of*, *d-extent-of*, *conf*) as follows.

- $\mathcal{I}_{\mathcal{CT}}$, $\mathcal{I}_{\mathcal{C}}$, and $\mathcal{I}_{\mathcal{D}}$ are sets that consist, respectively, of construct-type identifiers, construct identifiers, and construct-instance identifiers. We require $\mathcal{I}_{\mathcal{CT}}$, $\mathcal{I}_{\mathcal{C}}$, and $\mathcal{I}_{\mathcal{D}}$ to be pairwise disjoint sets.

- $\mathcal{V}$ is the set of values of construct and data identifiers.

- $val : \mathcal{I}_{\mathcal{C}} \cup \mathcal{I}_{\mathcal{D}} \rightarrow \mathcal{V}$ maps each construct and data identifier to a value.

- *ct-extent-of*: $\mathcal{I}_{\mathcal{CT}} \rightarrow \mathcal{P}(\mathcal{I}_{\mathcal{C}})$ maps construct-type identifiers to their associated construct-instance identifiers, where $\mathcal{P}(\mathcal{I}_{\mathcal{C}})$ denotes the powerset (set of subsets) of $\mathcal{I}_{\mathcal{C}}$.

- *c-extent-of*: $\mathcal{I}_{\mathcal{C}} \rightarrow \mathcal{P}(\mathcal{I}_{\mathcal{D}})$ maps construct identifiers to their associated construct-instance identifiers.

- *d-extent-of*: $\mathcal{I}_{\mathcal{D}} \rightarrow \mathcal{P}(\mathcal{I}_{\mathcal{D}})$ maps construct-instance identifiers to their associated construct-instance identifiers.

- *conf*: $\mathcal{I}_{\mathcal{C}} \times \mathcal{I}_{\mathcal{C}} \times \{*, +, ?, 1\} \times \{*, +, ?, 1\} \rightarrow$ *Boolean* is the conformance relation, such that given an input $<c, c', x, y>$ for $c, c' \in \mathcal{I}_{\mathcal{C}}$, *conf* returns true if c can conform to c' with domain constraint x and range constraint y.

The values $\mathcal{V}$ of $\mathcal{F}$ are represented as follows.

- The tuple $<(s_1, c_1), (s_2, c_2), ..., (s_n, c_n)>$ is in $\mathcal{V}$ for every construct identifier $c \in$ *ct-extent-of*(*struct-ct*) such that n is the number of components of c, s_i is the selector of the i-th component of c, and c_i is the construct identifier of the i-th component of c, for $1 \leq i \leq n$.

- The tuple $<(s_1, v_1), (s_2, v_2), ..., (s_n, v_n)>$ is in $\mathcal{V}$ for every construct-instance identifier $d \in \mathcal{I}_{\mathcal{D}}$ such that there is a $c \in$ *ct-extent-of*(*struct-ct*) where $d \in$ *c-extent-of*(c), and n is the number of components of d, s_i is the selector of the i-th component of d, and v_i is the value of the i-th component of d, for $1 \leq i \leq n$.

- The singleton c' is in $\mathcal{V}$ for every construct identifier $c \in$ *ct-extent-of*(*set-ct*) $\cup$ *ct-extent-of*(*list-ct*) $\cup$ *ct-extent-of*(*bag-ct*) such that c is defined as a collection of c'.

- The set $\{c_1, c_2, ..., c_n\}$ is in $\mathcal{V}$ for every union construct identifier $u \in$ *ct-extent-of*(*union-ct*) such that u is defined as a union of construct identifiers c_1, c_2, ..., c_n.

- The set $\{v_1, v_2, ..., v_n\}$ is in $\mathcal{V}$ for every construct-instance identifier $v \in$ *c-extent-of*(c) such that $c \in$ *ct-extent-of*(*set-ct*) and v is defined as the set of construct-instance identifiers v_1, v_2, ..., v_n.

- The bag $[v_1, v_2, ..., v_n]$ is in $\mathcal{V}$ for every construct-instance identifier $v \in$ *c-extent-of*(c) such that $c \in$ *ct-extent-of*(*bag-ct*) and v is defined as the bag of construct-instance identifiers v_1, v_2, ..., v_n.

- The tuple $<v_1, v_2, ..., v_n>$ is in $\mathcal{V}$ for every construct-instance identifier $v \in$ *c-extent-of*(c) such that $c \in$ *ct-extent-of*$($*list-ct*$)$ and v is defined as a list of construct-instance identifiers where v_1 is at index 1, v_2 is at index 2, ..., and v_n is at index n.

- The singleton v is in $\mathcal{V}$ for every construct-instance identifier $v \in$ *c-extent-of*(c) such that $c \in$ *ct-extent-of*$($*atomic-ct*$)$. Note that we assume an atomic value and its associated identifier are equivalent (in this case, v).

We assume that standard functions and relations are defined for the types of $\mathcal{V}$. That is, $v.s$ returns the corresponding construct or construct-instance identifier given a tuple v and a selector s (where s is a selector for v). The membership relation $\in$ is defined for sets, lists, and bags. The length-function $|v|$ takes a collection v and returns the number of elements of v. The function $v[i]$ returns the item at the ith position of v, where v is a list and i is a positive integer between 1 and $|v|$. And the function $x \in\in v$ returns the cardinality of x in v.

Finally, for a configuration $\mathcal{F}$, the interpretation $I_{\mathcal{F}}$ consists of a non-empty set Δ^I (the domain of the interpretation) and maps formulas in the query language to expressions in the configuration $\mathcal{F}$. We call $I_{\mathcal{F}}$ the *database instance mapping* of configuration $\mathcal{F}$. Note that we use the standard Datalog semantics, thus, we only describe the interpretation of atomic formulas for the purpose of determining truth assignments, and not arbitrary Datalog expressions. For example, the standard interpretation (for Datalog) of a relational database maps each constant to a constant in the domain of the database, and each formula $R(x_1, ..., x_n)$ for constants $x_1, ..., x_n$ to the corresponding tuple $< x_1, ..., x_n >$ of the relation instance R [73]. Given an interpretation $I_{\mathcal{F}}$ for the configuration $\mathcal{F}$, we define the domain of the interpretation Δ^I as follows. (Note that atomic values are given in $\mathcal{I}_{\mathcal{D}}$. Also, the second line adds component-selector definitions used in constructs, whereas the third line adds component selectors used in construct instances.)

$$\begin{aligned}\Delta^I = {} & \mathcal{I}_{\mathcal{CT}} \cup \mathcal{I}_{\mathcal{C}} \cup \mathcal{I}_{\mathcal{D}} \cup \\ & \{s \mid \exists c\ (c \in \textit{ct-extent-of}(\textit{struct-ct}) \wedge \textit{val}(c) = <..., (s, c'), ...>)\} \cup \\ & \{s \mid \exists d\ \exists c\ (d \in \textit{c-extent-of}(c) \wedge \textit{val}(d) = <..., (s, c'), ...>)\} \cup \\ & \{n \mid \exists d\ \exists c\ (d \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{set-ct}) \wedge |\textit{val}(d)| = n)\} \cup \\ & \{n \mid \exists d\ \exists c\ (d \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{list-ct}) \wedge |\textit{val}(d)| = n)\} \cup\end{aligned}$$

$$\{n \mid \exists d\ \exists c\ (d \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{bag-ct}) \wedge |val(d)| = n)\} \cup$$
$$\{i \mid \exists d\ \exists c\ (d \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{list-ct}) \wedge 1 \le i \le |val(d)|)\} \cup$$
$$\{i \mid \exists d\ \exists c\ (d \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{bag-ct}) \wedge 1 \le i \le |val(d)|)\}$$

The interpretation $I_{\mathcal{F}}$ is given in Figure 2.7 (for an arbitrary $\mathcal{F}$). Each expression gives a mapping such that the expression on the left side of the equality sign is true if and only if the expression on the right side of the equality sign is true.

$$\begin{array}{rcl}
\text{ct-inst}(x,\ y)^I & \equiv & x \in \textit{ct-extent-of}(y) \\
\text{c-inst}(x,\ y)^I & \equiv & x \in \textit{c-extent-of}(y) \\
\text{d-inst}(x,\ y)^I & \equiv & x \in \textit{d-extent-of}(y) \\
\text{conf}(x,\ y,\ d,\ r)^I & \equiv & \text{conf}(x,\ y,\ d,\ r) \\
(x.s\text{->}y)^I & \equiv & x \in \textit{ct-extent-of}(\textit{struct-ct}) \wedge val(x).s = y \\
(x.s{:}y)^I & \equiv & \exists(c)\ x \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{struct-ct}) \wedge val(x).s = y \\
(y \in x)^I & \equiv & \exists(c)\ x \in \textit{c-extent-of}(c) \wedge [c \in \textit{ct-extent-of}(\textit{set-ct}) \vee \\
 & & c \in \textit{ct-extent-of}(\textit{list-ct}) \vee c \in \textit{ct-extent-of}(\textit{bag-ct})] \wedge y \in val(x) \\
(x[i] = y)^I & \equiv & \exists(c)\ x \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{list-ct}) \wedge val(x)[i] = y \\
(|x| = n)^I & \equiv & \exists(c)\ x \in \textit{c-extent-of}(c) \wedge [c \in \textit{ct-extent-of}(\textit{set-ct}) \vee \\
 & & c \in \textit{ct-extent-of}(\textit{list-ct}) \vee c \in \textit{ct-extent-of}(\textit{bag-ct})] \wedge |val(x)| = n \\
(y \in\in x = n)^I & \equiv & \exists(c)\ x \in \textit{c-extent-of}(c) \wedge c \in \textit{ct-extent-of}(\textit{list-ct}) \wedge y \in\in val(x) = n \\
\text{setof}(x,\ y)^I & \equiv & x \in \textit{ct-extent-of}(\textit{set-ct}) \wedge val(x) = y \\
\text{bagof}(x,\ y)^I & \equiv & x \in \textit{ct-extent-of}(\textit{bag-ct}) \wedge val(x) = y \\
\text{listof}(x,\ y)^I & \equiv & x \in \textit{ct-extent-of}(\textit{list-ct}) \wedge val(x) = y \\
\text{unionof}(x,\ y)^I & \equiv & x \in \textit{ct-extent-of}(\textit{union-ct}) \wedge \text{y} \in val(x)
\end{array}$$

Figure 2.7: The interpretation of ULD configurations.

2.5 The ULD Constraint Language

The ability to query the ULD is important for providing uniform access to information sources. The ULD query language can also be used to specify additional constraints on configurations. In this section, we describe a special syntax for expressing constraints based on the ULD query language. Our goal is to define a language that can be used to describe common ULD constraints, while exploiting the expressive power of Datalog. Therefore, we have chosen a syntax for expressing ULD constraints that can be directly converted to stratified Datalog$\neg$ programs. We also show how such constraints can be used to further refine data models, and in particular, to express conformance constraints beyond simple cardinality restrictions. In general, further refining a data model by providing additional constraints using the language presented here is optional. However, we require domain and range cardinality restrictions on conformance, for example, so that additional tools can determine whether schema is required or optional.

Constraints against the ULD are represented as stratified Datalog¬ programs with a specific set of formulas. The result of executing the program against a configuration is used to determine whether the constraint is satisfied. To make constraints easier to specify, we introduce the following syntax for defining ULD constraints, where the name of the constraint is represented as t, $n \geq 1$, and $m \geq 0$.

$$
\begin{array}{l}
\textsf{constraint } t\text{:} \\
\quad e_1, e_2, ..., e_n \Leftarrow b_c. \\
\qquad\qquad r_1 \leftarrow b_1. \\
\qquad\qquad r_2 \leftarrow b_2. \\
\qquad\qquad \vdots \\
\qquad\qquad r_m \leftarrow b_m.
\end{array}
$$

A constraint expressed with this syntax can be directly converted to a corresponding stratified Datalog¬ program, which we show below. Due to the underlying conversion, we require that t-body and t-sat (which are introduced by the conversion) are not used as predicate names in r_1 to r_m below, where t is replaced by the name of the constraint. All ULD constraints consist of exactly one rule called the *constraint assertion*, plus an optional set of rules forming a stratified Datalog¬ program, called the *constraint program*. The constraint assertion is shown as the first rule above and the remaining rules define the constraint program. As shown, constraint-assertion rules are distinguished from constraint-program rules by the $\Leftarrow$ implication symbol. The head of a constraint assertion has one or more extensional relations e_1 to e_n, and a standard stratified Datalog¬ rule body b_c containing formulas with either extensional relations or intensional relations taken from r_1 to r_m. Unlike Datalog, the head of a constraint assertion can contain multiple formulas, positive formulas with variables not mentioned in the body of the rule, and negative formulas. We require all variables occurring in a negative formula in the head of an assertion to occur in a positive formula in the assertion (either the head or the body).

A constraint assertion serves to define the purpose of the constraint, and the constraint program serves to define temporary, intensional relations used by the constraint assertion. The meaning of a constraint assertion is defined as follows. Let v be a function that maps a set of variable names to variable assignments (that is, values) and let var be a function that takes one or more formulas and returns the corresponding set of variables. The constraint program is executed first. The body b_c of the constraint assertion is then evaluated against both the configuration facts and the resulting IDB, generated from executing the constraint program. Each evaluation of b_c results in a variable assignment v_b that makes b_c true with respect to the configuration and the IDB. Given a formula e, we write e/v to denote a new formula with every variable in e replaced by its corresponding value in the variable assignment v. Similarly, for variable assignments v_1 and v_2, $e/v_1/v_2$ denotes the formula with every variable in e/v_1 replaced by its corresponding value in the variable assignment v_2, and so on. Given a configuration $\mathcal{F}$ with corresponding database instance mapping $I_{\mathcal{F}}$ and constraint t, we use the standard notation $I_{\mathcal{F}} \models t$ whenever $I_{\mathcal{F}}$ satisfies constraint t. For a configuration instance $I_{\mathcal{F}}$ and constraint t, $I_{\mathcal{F}} \models t$ if and only if, for each variable assignment v_b of b_c, the following conditions are true.

1. If e_i in e_1, e_2, ..., e_n is a positive formula with every variable in $var(e_i)$ occurring in $var(b_c)$ (that is, $var(e_i) \subseteq var(b_c)$), then $e_i/v_b \in I_{\mathcal{F}}$.

2. If e_i in e_1, e_2, ..., e_n is a negative formula of the form $\neg f_i$ with every variable in $var(f_i)$ occurring in $var(b_c)$ (that is, $var(f_i) \subseteq var(b_c)$), then $f_i/v_b \notin I_{\mathcal{F}}$.

3. If E is the largest subset of formulas e_1 to e_n such that every $e_j \in$ E has at least one variable in $var(e_j)$ not occurring in $var(b_c)$ (that is, $\exists$V (V $\in var(e_j) \wedge$ V $\notin var(b_c)$)), then there must exist at least one variable assignment v_e for E that makes each formula in E true for the configuration. That is, for $E = \{e_{j1}, e_{j2}, ..., e_{jk}\}$ and $var(E) - var(b_c) = \{$V$_1$,V$_2$,...,V$_l\}$, the set E represents the first-order logic expression $\exists$V$_1$ $\exists$V$_2$ ... $\exists$V$_l$ $(e_{j1}/v_b, e_{j2}/v_b, ..., e_{jk}/v_b)$. A variable assignment v_e makes E true with respect to the configuration if and only if for every formula $e_j \in E$, $e_j/v_b/v_e \in I_{\mathcal{F}}$. (Note that according to the definition of the ULD constraint syntax, E can only contain positive formulas.)

The ULD constraint language is meant to simplify the task of expressing constraints against ULD configurations. Every well-formed ULD constraint can be rewritten as a stratified Datalog¬ program as follows, where t is the name of the constraint and both t-body and t-sat are intensional relations, introduced as part of the re-writing process.

t-sat$(var(b_c)) \leftarrow t$-body$(var(b_c))$, e_1, e_2, ..., e_n.
t-body$(var(b_c)) \leftarrow b_c$.
$r_1 \leftarrow b_1$.
$r_2 \leftarrow b_2$.
$\vdots$
$r_m \leftarrow b_m$.

Note that we use the syntax $p(var(b_c))$ to denote a formula $p(V_1, V_2, ...)$ where $var(b_c) = \{V_1, V_2, ...\}$. If $var(b_c)$ is empty (that is, there are no variables in the body of the constraint assertion), we substitute $var(b_c)$ with an arbitrary constant symbol (for example, we could choose a constant from one of the formulas of b_c). The use of a constant symbol is needed because Datalog requires at least one attribute in every relation.

With this Datalog¬ representation of a ULD constraint, it is straightforward to check whether the constraint is satisfied by a configuration (note that we assume configurations are constrained to be finite). Namely, $I_{\mathcal{F}} \models t$ if and only if for each fact in the evaluation of the formula t-sat there is a corresponding fact in the evaluation of t-body. That is, for each tuple of t-sat there is an identical tuple in t-body and vice versa. For example, consider the following constraint (Example 7), which states that, within an RDF configuration, no two classes may have the same label.

Example 7 (*RDF class-label constraint*)

constraint key1:
    C1=C2 ⇐ c-inst(C1, class), c-inst(C2, class), C1.hasLabel:L, C2.hasLabel:L.

The RDF class-label constraint can be directly re-written into the following Datalog rules.

t-sat(C1, C2, L) ← t-body(C1, C2, L), C1=C2.

t-body(C1, C2, L) ← c-inst(C1, class), c-inst(C2, class), C1.hasLabel:L, C2.hasLabel:L.

Assume a configuration that contains two distinct data identifiers, say d_1 and d_2, that are both classes with the same label l. Thus, the formula t-body(d_1,d_2,l) is true. However, there is not an equivalent t-sat formula: t-sat(d_1,d_2,l) is false because d_1 does not equal d_2 (the C1=C2 expression is false).

We show below that the two representations for constraints are equivalent, that is, constraints expressed in the abbreviated syntax and the evaluation of the stratified Datalog¬ programs they generate. We start by showing that the generated stratified Datalog¬ syntax is well-formed.

Theorem 1 (Well-formed Datalog-program generation) *Given a valid ULD constraint t, the Datalog program $\mathcal{D}$ generated from the constraint is a well-formed stratified Datalog¬ program.*

Proof (Well-formed Datalog-program generation). To show that $\mathcal{D}$ can be stratified, notice that t's constraint program is by definition a valid stratified Datalog¬ program (because *t-body* and *t-sat* are not allowed as intensional predicate names for the IDB formulas r_1 to r_m). Assume P_1, P_2, ..., P_i are used to partition the rules from t's constraint program, where i is the number of needed partitions. The rule for *t-body* is placed in partition P_{i+1}, because b_c may contain negative IDB (constraint program) formulas. The rule for *t-sat* is also placed in P_{i+1} because it contains a positive (not a negative) *t-body* formula. While *t-sat* can contain negative formulas, they are restricted to extensional formulas only.

Finally, *t-sat* and *t-body* rules are valid Datalog¬ rules as follows. By definition, b_c must be a valid Datalog¬ rule body. If $var(b_c)$ is empty, *t-body* is still well-formed because a constant term is added to the formula. For non-empty $var(b_c)$, every variable in the head of the *t-sat* rule also occurs in the body of the rule. The head of a constraint assertion may only contain a negative formula if each variable in the formula occurs in a positive formula in the head or body of the rule. Note that all variables that occur in the head of the assertion occur in the body of the *t-sat* rule. Therefore, the *t-sat* rule satisfies the Datalog¬ restriction concerning negative formulas: For a given negative formula e_i in an assertion, each variable of e_i either occurs in a positive formula e_j in the body of *t-sat*, or in the positive *t-body* formula in the body of *t-sat*. *QED*

Finally, we show that the two representations for constraints are semantically equivalent. That is, if t' is the constraint defined by the stratified Datalog¬ program $\mathcal{D}$ and t is a valid ULD constraint, we say that t and $\mathcal{D}$ are *semantically equivalent* if for any configuration $\mathcal{F}$ with corresponding database instance mapping $I_{\mathcal{F}}$, $I_{\mathcal{F}} \models t$ if and only if $I_{\mathcal{F}} \models t'$.

Theorem 2 (Semantic equivalence of Datalog-program generation) *Given any valid ULD constraint t and Datalog program $\mathcal{D}$ generated from t, $\mathcal{D}$ and t are semantically equivalent.*

Proof (Semantic equivalence of Datalog-program generation). Assume that $\mathcal{F}$ is a given configuration with database instance mapping $I_{\mathcal{F}}$. A variable assignment v_b makes b_c true if and only if v_b makes *t-body* true. For a given variable assignment v_b that makes b_c true, *t-sat* is true for v_b if and only if the expression e_1, e_2, ..., e_n is true. It follows that $I_{\mathcal{F}} \models t'$, where t' represents the constraint imposed by $\mathcal{D}$, if only if for every v_b that makes b_c true, *t-sat* is also true. (Note that by definition of the rewriting, t' and t have the same b_c and e_1 to e_n.) Thus, $I_{\mathcal{F}} \models t'$ if and only if each e_i/v_b is true for $1 \leq i \leq n$, which is exactly equivalent to the conditions given by the ULD constraint definition for determining whether $I_{\mathcal{F}} \models t$ is true. That is, e_i/v_b is true for $1 \leq i \leq n$ if and only if the three conditions given by the ULD constraint definition hold. *QED*

The rest of this section gives examples of the ULD constraint language. The following constraint (Example 8) uses the *subclassClosure* (intensional) relation to ensure all RDF objects are instances of their corresponding classes (which may not be explicitly represented in an RDF source, but is usually inferred by an RDF processor). Note that the *subclassClosure* relation computes the transitive closure of the RDF subclass relationship.

Example 8 (*RDF(S) subclass instance constraint*)

constraint obj1:
  d-inst(R, C2) $\Leftarrow$ c-inst(R, resource), d-inst(R, C1), subclassClosure(C1, C2).

  subclassClosure(C1, C2) $\leftarrow$ C1=C2.
  subclassClosure(C1, C3) $\leftarrow$ c-inst(S, subclass), S.hasSub:C1,
    S.hasSuper:C2, subclassClosure(C2, C3).

In general, constraints express information concerning configurations that are not directly expressible using the available structures of the ULD. Constraints can be used for a number of purposes such as validating that configurations (or the result of updating configurations) are correct, optimizing queries [2, 67], and transformation [68].

An important use of constraints for the ULD is to further define conformance relationships. To illustrate, Example 9 describes some of the restrictions imposed by conformance relationships in the relational model of Example 1. Note that in the relational model definition of Example 1, the conformance relationships only require tables and tuples to conform to relation types (through cardinality constraints) but they do not describe the structural implications of the conformance relationships. The constraints of Example 9 capture some of these structural requirements.

Example 9 (*Relational conformance constraints*)

constraint rel-conf1:
  d-inst(T, R) $\Leftarrow$ c-inst(B, table), c-inst(R, relation), d-inst(B, R), T $\in$ B.
constraint rel-conf2:
  d-inst(V, D) $\Leftarrow$ c-inst(T, tuple), c-inst(R, relation), d-inst(T, R), R.hasAtts:AL,
    AL[I]=A, A.hasDomain:D, T[I]=V.
constraint rel-conf3:
  |T|=N $\Leftarrow$ c-inst(T, tuple), c-inst(R, relation), d-inst(T, R), R.hasAtts:AL, |AL|=N.

The first constraint (rel-conf1) states that, for a table to conform to a relation, each tuple in the table must be a *d-inst* of the relation type. The second and third constraints define the conformance relationship between tuples and relations. The second constraint (rel-conf2) requires the *i*-th value in the tuple to be a *d-inst* of the domain of the relation's

i-th attribute. The last constraint (rel-conf3) states that a tuple must have the same number of values as the number of attributes in the relation type.

Example 10 defines a conformance constraint for the RDF(S) data model of Example 3. Note that RDF(S) places few restrictions on conformance, for example, objects are not required to have any properties of their corresponding classes (that is, properties are optional). The constraint below, however, states that, if a triple's predicate is a defined property, the triple's subject must be a *d-inst* of the property's domain class, and the triple's object must be a *d-inst* of the property's range class. Note that this is an implicit conformance constraint, because there is not an explicit conformance definition between triples and properties.

Example 10 (*RDF(S) conformance constraint*)

constraint rdf-conf1:
d-inst(R_s, C_s), d-inst(R_o, C_o) $\Leftarrow$
c-inst(T, triple), c-inst(P, property), T.hasPred:P, T.hasSubj=R_s,
P.hasDomain=C_s, T.hasObj:R_o, P.hasRange:C_o.

Not all constraints can be expressed using the ULD constraint language described in this section. For example, it is not possible to derive a disjunction, in which the intent of the constraint is to ensure that either e_1 *or* e_2 must hold. It is also not possible to express constraints that cannot be stratified. The ULD constraint language is slightly more expressive than stratified Datalog$\neg$ programs, because it permits existentially quantified and negative formulas in the head of an assertion. We show in Chapter 3 that the ULD constraint language is expressive enough to capture a wide range of data-model constraints.

2.6 A Partial ULD Axiomatization

The ULD provides uniform access to data model, schema, and data information. This flexibility allows generic restrictions to be placed on configurations using the ULD constraint language defined in the previous section. In other words, it is possible to define general constraints (applicable to any data model) that must be satisfied by all valid configurations using the ULD constraint language. This section exploits this observation to further formalize the ULD framework. In particular, we provide an axiomatization for the ULD using the ULD constraint language. By *axiomatization*, we mean a set of statements (*axioms*) that must be true for all configurations. If an axiom is false for a configuration, the configuration is not considered valid. That is, the configuration is not a model of the axioms. Note that the axiomatization is independent of the syntax used to represent a configuration, and we assume that all configurations are syntactically valid. The axiomatization we present in this section is partial, and attempts to characterize the description of the ULD given previously in Sections 2.1, 2.2, and 2.3. There are true statements that can be made of valid configurations that are not included in the axiomatization. First, not all true statements can be expressed using the ULD constraint language. Examples include axioms for equality, collection arithmetic (such as asserting that the length of a collection is always the same as the the number of elements in the collection), and some closure axioms that require disjunction in the head of a rule. Second, we do not include all "trivial" axioms (mainly for conciseness), for example, that a domain constraint can only contain a value in the set {1,+,?,*}.

We divide the axiomatization into two parts. First, we give a core set of axioms that are true regardless of the meta-data-model chosen. And second, we give the specific axioms for the ULD meta-data-model (with tuple, collection, union, and atomic structures). Although we do not consider alternative meta-data-models, it is possible to define different structures and constraints by choosing different axioms than those given for the ULD meta-data-model.[6]

When defining axioms, we use the conventions for variable names as shown in Figure 2.8. That is, we use variables S and T for construct types, P and Q for constructs, and X and Y for construct instances. When appropriate, we use the convention that P is a *ct-inst* of S, Q is a *ct-inst* of T, P can conform to Q, X is a *c-inst* of P, Y is a *c-inst* of Q, and X is a *d-inst* of Y. We also use the term axiom instead of constraint to define the

[6]In addition to defining an alternative meta-data-model, the syntax used to specify configurations would also need to be altered along with the mapping from the syntax to the corresponding database instance.

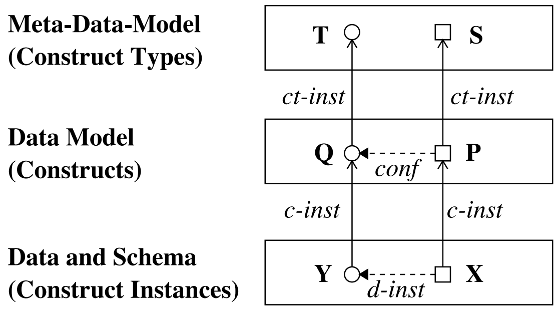

Figure 2.8: The naming conventions used in specifying ULD axioms.

rules in the theory. Semantically, constraints and axioms are identical, however, we use constraint to represent restrictions placed on specific data models and axiom to represent restrictions placed over all configurations regardless of the data model.

2.6.1 Definition of the Core ULD Theory

The set of axioms shown in Figure 2.9 and 2.10 are independent of any particular ULD meta-data-model. The axioms define the *core theory* of the ULD, which performs three primary functions: (1) it describes the dependency among construct types, constructs, and construct instances, (2) it partitions construct types, constructs, and construct instances into distinct sets, and (3) it articulates the relationship between conformance specifications (*conf* facts) and data instance-of relationships (*d-inst* facts). Note that we use the symbol * to distinguish axioms that follow directly from the interpretation given in Section 2.4.2.

The first three axioms of Figure 2.9 describe the dependencies between construct types, constructs, and construct instances. The first axiom requires every construct identifier used in a configuration to be a *ct-inst* of some construct type. Recall that the $\Leftarrow$ symbol denotes a constraint assertion in which existential variables are permitted in the head of the rule. Thus, the formula ct-inst(P, S) requires at least one S for each P in the configuration, where the equivalent first-order logic expression is: $\forall$X $\forall$P (c-inst(X, P) $\Rightarrow$ $\exists$S ct-inst(P, S)). The second axiom requires the participating identifiers of every *d-inst* relationship to have constructs that occur together in a conformance definition. The third axiom requires conformance definitions to relate constructs (as opposed to construct types

axiom 1: (*Constructs always have construct types.*)
  ct-inst(P, S) ⇐ c-inst(X, P).

axiom 2: (*Data instances always have conforming constructs.*)
  c-inst(X, P), c-inst(Y, Q), conf(P, Q, D, R) ⇐ d-inst(X, Y).

axiom 3*: (*Conformance is defined between constructs.*)
  ct-inst(P, S), ct-inst(Q, T) ⇐ conf(P, Q, D, R).

axiom 4*: (*Construct and construct types are disjoint.*)
  S≠Q ⇐ ct-inst(P, S), ct-inst(Q, T).

axiom 5*: (*Constructs and construct instances are disjoint.*)
  P≠Y ⇐ ct-inst(P, S), c-inst(Y, Q).

axiom 6*: (*Construct types and construct instances are disjoint.*)
  S≠Y ⇐ ct-inst(P, S), c-inst(Y, Q).

axiom 7: (*Domain and Range constraints are unique among conforming constructs*)
  D1=D2, R1=R2 ⇐ conf(P, Q, D1, R1), conf(P, Q, D2, R2).

Figure 2.9: The core ULD axioms for describing basic structure restrictions.

or construct instances).

Axioms 4 through 6 give pairwise restrictions on ULD meta-data-model identifiers. Axiom 4 requires the set of construct types and constructs to be disjoint. Axiom 5 requires the set of constructs and construct instances to be disjoint. And Axiom 6 requires the set of construct types and construct instances to be disjoint.

Axiom 7 requires unique conformance definitions, that is, if P can conform to Q, there is only one such definition for P and Q in a configuration. Note that conformance is not a symmetric relationship. However, it is possible to define separate conformance relationships such that P can conform to Q and Q can conform to P (but asserting P conforms to Q does not imply Q conforms to P).

The remaining axioms of the core theory are shown in Figure 2.10 and define the domain and range cardinality restrictions on conformance. Recall that a conformance definition takes the form *conf*(P, Q, D, R), which states that an instance X of P can conform to an instance Y of Q (that is, there can be a d-inst(X, Y) fact in the configuration). In

axiom 8: (*Range constraint requiring at least one conformee.*)
c-inst(Y, Q), d-inst(X, Y) $\Leftarrow$ conf(P, Q, D, 1), c-inst(X, P).

axiom 9: (*Domain constraint requiring at least one conformer.*)
c-inst(X, P), d-inst(X, Y) $\Leftarrow$ conf(P, Q, 1, D), c-inst(Y, Q).

axiom 10: (*Range constraint requiring at least one conformee.*)
c-inst(Y, Q), d-inst(X, Y) $\Leftarrow$ conf(P, Q, D, +), c-inst(X, P).

axiom 11: (*Domain constraint requiring at least one conformer.*)
c-inst(X, P), d-inst(X, Y) $\Leftarrow$ conf(P, Q, +, D), c-inst(Y, Q).

axiom 12: (*Range constraint requiring at most one conformee.*)
Y_1=Y_2 $\Leftarrow$ conf(P, Q, D, 1), c-inst(X, P), c-inst(Y_1, Q), c-inst(Y_2, Q), d-inst(X, Y_1), d-inst(X, Y_2).

axiom 13: (*Domain constraint requiring at most one conformer.*)
X_1=X_2 $\Leftarrow$ conf(P, Q, 1, R), c-inst(Y, Q), c-inst(X_1, P), c-inst(X_2, P), d-inst(X_1, Y), d-inst(X_2, Y).

axiom 14: (*Range constraint requiring at most one conformee.*)
Y_1=Y_2 $\Leftarrow$ conf(P, Q, D, ?), c-inst(X, P), c-inst(Y_1, Q), c-inst(Y_2, Q), d-inst(X, Y_1), d-inst(X, Y_2).

axiom 15: (*Domain constraint requiring at most one conformer.*)
X_1=X_2 $\Leftarrow$ conf(P, Q, ?, R), c-inst(Y, Q), c-inst(X_1, P), c-inst(X_2, P), d-inst(X_1, Y), d-inst(X_2, Y).

Figure 2.10: The core ULD axioms describing conformance cardinality restrictions.

addition, D represents a constraint on the domain (that is, the P side) of the relationship, and R represents a constraint on the range (that is, the Q side) of the relationship. For example, if the range cardinality restriction is 1, every instance X of P must conform to some instance Y of Q (that is, $\forall$X $\exists$Y (d-inst(X, Y))). Similarly, if the domain cardinality restriction is 1, there must exist an instance X of P for every instance Y of Q such that X conforms to Y (that is, $\forall$Y $\exists$X (d-inst(X, Y))).

We note that the core theory is less restrictive structurally than the ULD meta-data-model. In particular, the core theory permits a construct to be an instance of multiple construct types and to conform to multiple constructs.

axiom 16*: (*Struct construct-type introduction.*)
S=struct-ct ⇐ ct-inst(P, S), ¬ S=set-ct, ¬ S=bag-ct, ¬ S=list-ct, ¬ S=atomic-ct.

axiom 17*: (*Set construct-type introduction.*)
S=set-ct ⇐ ct-inst(P, S), ¬ S=struct-ct, ¬ S=bag-ct, ¬ S=list-ct, ¬ S=atomic-ct.

axiom 18*: (*Bag construct-type introduction.*)
S=bag-ct ⇐ ct-inst(P, S), ¬ S=struct-ct, ¬ S=set-ct, ¬ S=list-ct, ¬ S=atomic-ct.

axiom 19*: (*List construct-type introduction.*)
S=list-ct ⇐ ct-inst(P, S), ¬ S=struct-ct, ¬ S=set-ct, ¬ S=bag-ct, ¬ S=atomic-ct.

axiom 20*: (*Atomic construct-type introduction.*)
S=atomic-ct ⇐ ct-inst(P, S), ¬ S=struct-ct, ¬ S=set-ct, ¬ S=bag-ct, ¬ S=list-ct.

axiom 21: (*A construct is an instance of at most one construct type.*)
S_1=S_2 ⇐ ct-inst(P, S_1), ct-inst(P, S_2).

axiom 22: (*A construct conforms to at most one construct.*)
Q_1=Q_2 ⇐ conf(P, Q_1, D_1, R_1), conf(P, Q_2, D_2, R_2).

Figure 2.11: The ULD meta-data-model axioms that define the allowable construct types and two structural restrictions.

2.6.2 Definition of the ULD Meta-Data-Model Theory

The ULD meta-data-model axioms are given in Figures 2.11–2.17. The meta-data-model axioms extend the core theory by defining the allowable construct types and the corresponding representations for basic structures.

The first five axioms (16–22) of Figure 2.11 restrict the set of construct-type constants to *struct-ct*, *set-ct*, *bag-ct*, *list-ct*, and *atomic-ct*. Thus, any configuration that uses a different construct type than those defined is not considered valid. Taken together, Axioms 16—20 are equivalent to the first-order logic expression ∀P ∀S (ct-inst(P, S) ⇒ S=struct-ct ∨ S=set-ct ∨ S=bag-ct ∨ S=list-ct ∨ S=atomic-ct ∨ S=union-ct), however, disjunction constraints are not allowed directly. The last two axioms (20 and 21) define structural restrictions on configurations. In particular, Axiom 20 restricts a construct from being a *ct-inst* of more than one construct type, and Axiom 22 refines Axiom 7 of the core theory by restricting a construct to conform to at most one other construct.

Figure 2.12 defines axioms for tuple structures (that is, *struct-ct* basic structures). The

axiom 23*: (*Component definitions exist only for struct constructs.*)
ct-inst(P, struct-ct), ct-inst(Q, T) $\Leftarrow$ P.Sel->Q.

axiom 24*: (*Component expressions exist only for struct construct instances.*)
c-inst(X, P), ct-inst(P, struct-ct), c-inst(Y, Q) $\Leftarrow$ X.Sel:Y.

axiom 25: (*Every struct construct has at least one component definition.*)
P.Sel->Q $\Leftarrow$ ct-inst(P, struct-ct).

axiom 26: (*Struct instances always have a component definition for each component.*)
X.Sel:Y, c-inst(Y, Q) $\Leftarrow$ ct-inst(P, struct-ct), P.Sel->Q, c-inst(X,P).

axiom 27: (*Struct instances always have components for component definitions.*)
P.Sel->Q, c-inst(Y, Q) $\Leftarrow$ ct-inst(P, struct-ct), c-inst(X, P), X.Sel:Y.

axiom 28: (*Struct constructs have unique selectors.*)
Q_1=Q_2 $\Leftarrow$ ct-inst(P, struct-ct), P.Sel->Q_1, P.Sel->Q_2.

axiom 29: (*Struct instances have unique selectors.*)
Y_1=Y_2 $\Leftarrow$ ct-inst(P, struct-ct), c-inst(X, P), P.Sel:Y_1, P.Sel:Y_2.

Figure 2.12: The axioms for tuple structures in the ULD meta-data-model.

first two axioms (23 and 24) define component expressions. For example, Axiom 23 requires P to be a *struct-ct* construct and Q to be a construct in a component definition P.Sel->Q. Axiom 25 requires *struct-ct* constructs to have at least one component definition. Axioms 26 and 27 require each construct to have exactly one component for each corresponding component definition. And Axioms 28 and 29 require *struct-ct* constructs and construct instances to have unique component selectors.

Figure 2.13 defines axioms for collections (that is, *set-ct*, *bag-ct*, and *list-ct* basic structures). Axiom 30 requires that membership occurs between construct instances and that membership formulas are applied only to collections. Axiom 31 defines the same restrictions as Axiom 30 for collection-length formulas.

Figures 2.14–2.16 axiomatize the set, bag, and list structures, respectively. Axioms 32, 37, and 44 require collection definitions to occur only for collection constructs. Axioms 33, 38, and 45 require every collection construct to have a collection definition. Axioms 34, 39, and 46 require a collection construct to have at most one collection definition. Axioms 35, 40, and 47 require collection instances to consist of only valid members, that is, members

axiom 30*: (*The membership predicate only applies to collection construct instances.*)
c-inst(X, P), c-inst(Y, Q), $\neg$ ct-inst(P, struct-ct), $\neg$ ct-inst(P, atomic-ct)$\Leftarrow$ Y $\in$ X.

axiom 31*: (*Only collections have an element count.*)
c-inst(X, P), $\neg$ ct-inst(P, struct-ct), $\neg$ ct-inst(P, atomic-ct) $\Leftarrow$ |X|=C.

Figure 2.13: Two general axioms for collection structures in the ULD meta-data-model.

axiom 32*: (*Set definitions are between a set construct and another construct.*)
ct-inst(P, set-ct), ct-inst(Q, T) $\Leftarrow$ setof(P, Q).

axiom 33*: (*Set constructs always have a set definition.*)
setof(P, Q) $\Leftarrow$ ct-inst(P, set-ct).

axiom 34: (*Set constructs have at most one set definition.*)
Q_1=Q_2 $\Leftarrow$ ct-inst(P, set-ct), setof(P, Q_1), setof(P, Q_2).

axiom 35: (*Set instances always have valid members.*)
c-inst(Y, Q) $\Leftarrow$ ct-inst(P, set-ct), setof(P, Q), c-inst(X, P), Y$\in$X.

axiom 36*: (*Set instances always have a length.*)
|X|=C $\Leftarrow$ ct-inst(P, set-ct), c-inst(X, P).

Figure 2.14: The axioms for set structures in the ULD meta-data-model.

that are instances of the construct mentioned in the collection definition. And axioms 36, 41, and 48 require collection instances to have a cardinality.

Axioms 42 and 43 of Figure 2.15 define the duplicate-element predicate for bags. Recall that for the bag x, construct-instance y, and integer c, the expression $y \in\in x = c$ states that y occurs c times as a member of x. Axiom 42 requires each member of a bag to have an associated duplicate-element formula. And Axiom 43 limits duplicate-element formulas to bag construct instances.

Axioms 49 and 50 of Figure 2.16 define the list-index predicate. Axiom 49 requires each member in a list to have an associated index in the list. And Axiom 50 limits list-index expressions to members of list construct instances.

Figure 2.17 gives the axioms for unions (that is, *union-ct* constructs). Axioms 51

axiom 37*: (*Bag definitions are between a bag construct and another construct.*)
ct-inst(P, bag-ct), ct-inst(Q, T) $\Leftarrow$ bagof(P, Q).

axiom 38*: (*Bag constructs always have a bag definition.*)
bagof(P, Q) $\Leftarrow$ ct-inst(P, bag-ct).

axiom 39: (*Bag constructs have at most one bag definition.*)
$Q_1 = Q_2$ $\Leftarrow$ ct-inst(P, bag-ct), bagof(P, Q_1), bagof(P, Q_2).

axiom 40*: (*Bag instances always have valid members.*)
c-inst(Y, Q) $\Leftarrow$ ct-inst(P, bag-ct), bagof(P, Q), c-inst(X, P), $Y \in X$.

axiom 41*: (*Bag instances always have a length.*)
$|X| = C$ $\Leftarrow$ ct-inst(P, bag-ct), c-inst(X, P).

axiom 42*: (*Bag instances always have a duplicate-element expression for each member.*)
$Y \in \in X = C$ $\Leftarrow$ ct-inst(P, bag-ct), c-inst(X, P), $Y \in X$.

axiom 43*: (*Only bags have duplicate-element expressions.*)
c-inst(X, P), ct-inst(P, bag-ct), c-inst(Y, Q) $\Leftarrow$ $Y \in \in X = C$.

Figure 2.15: The axioms for bag structures in the ULD meta-data-model.

and 52 require that every union definition is for a union construct (Axiom 51) and union constructs have at least one union definition (Axiom 52). Axiom 53 requires each instance of a union construct to also be an instance of another, non-union construct. Axiom 54 requires each instance of a construct that participates in a union construct to be an instance of the union construct. Axiom 55 disallows union constructs from conforming to any other construct. And Axiom 56 restricts union definitions from being acyclic.

Besides serving as a formal description of the ULD and the ULD meta-data-model, the axiomatization described in this section can be used to (partially) validate a configuration directly. That is, by using the ULD constraint language to express axioms, we could execute these axioms using a Datalog or Prolog system to check that a configuration is valid.

In addition, we believe the axiomatization can be used to infer new facts concerning configurations and to derive additional statements, for further reasoning. One use of the axiomatization is to automate the population of a configuration, for example, when we

axiom 44*: (*List definitions are between a list construct and another construct.*)
ct-inst(P, list-ct), ct-inst(Q, T) $\Leftarrow$ listof(P, Q).

axiom 45*: (*List constructs always have list definitions.*)
listof(P, Q) $\Leftarrow$ ct-inst(P, list-ct).

axiom 46: (*List constructs have at most one definition.*)
$Q_1=Q_2$ $\Leftarrow$ ct-inst(P, list-ct), listof(P, Q_1), listof(P, Q_2).

axiom 47*: (*List instances have only valid members.*)
c-inst(Y, Q) $\Leftarrow$ ct-inst(P, list-ct), listof(P, Q), c-inst(X, P), $Y \in X$.

axiom 48*: (*List instances always have a length.*)
$|X|=C$ $\Leftarrow$ ct-inst(P, list-ct), c-inst(X, P).

axiom 49*: (*List instances always have an index expression for each member.*)
$X[I]=Y$ $\Leftarrow$ ct-inst(P, list-ct), c-inst(X, P), $Y \in X$.

axiom 50*: (*Only lists have indexes.*)
c-inst(X, P), ct-inst(P, list-ct), c-inst(Y, Q) $\Leftarrow$ $X[I]=Y$.

Figure 2.16: The axioms for list structures in the ULD meta-data-model.

wish to transform one configuration into another configuration or an information source into its corresponding configuration. We describe this use of constraints in more detail in Chapter 4.

As a simple example, assume we have defined a rule to convert existing information into an RDF resource (using the RDF(S) data model defined in Example 3). However, we would like to know if an instance of an RDF resource must be an instance of any other construct (so our program can define a valid configuration). This question can be answered using Axiom 54, which states that any instance of a construct that participates in a union definition is also an instance of the union construct. Therefore, we can use Axiom 54 to prove that every `resource` construct instance must also be an `objVal` constant instance.

As another example, using Axioms 2 and 22 we can derive the following true statement concerning configurations.

c-inst(P, Q) $\Leftarrow$ d-inst(X, Y), c-inst(X, P), conf(P, Q, D, R).

This axiom allows us to infer the construct of an instance that participates in a *d-inst*

axiom 51*: (*Union definitions exist only for union constructs.*)
ct-inst(P, union-ct), ct-inst(Q, T) $\Leftarrow$ unionof(P, Q).

axiom 52: (*Union constructs are composed of at least two distinct constructs.*)
unionof(P, Q_1), unionof(P, Q_2), $\neg$ Q_1=Q_2, $\Leftarrow$ ct-inst(P, union-ct).

axiom 53: (*Union constructs do not have direct instances.*)
c-inst(X, Q), $\neg$ ct-inst(Q, union-ct) $\Leftarrow$ ct-inst(P, union-ct), c-inst(X, P).

axiom 54*: (*Instances of constructs grouped by union are instances of the union.*)
c-inst(Y, P) $\Leftarrow$ ct-inst(P, union-ct), unionof(P, Q), c-inst(Y, Q).

axiom 55: (*Union constructs do not conform to other constructs.*)
$\neg$ conf(P, Q, D, R) $\Leftarrow$ ct-inst(P, union-ct), ct-inst(Q, T), conf(P′, Q′, D, R).

axiom 56: (*Union constructs are acyclic.*)
P$\neq$P′ $\Leftarrow$ c-inst(P, union-ct), unionConnected(P, P′).

unionConnected(P_1, P_2) $\leftarrow$ c-inst(P_1, union-ct), unionof(P_1, P_2).
unionConnected(P_1, P_3) $\leftarrow$ c-inst(P_1, union-ct), unionof(P_1, P_2), unionConnected(P_2, P_3).

Figure 2.17: The axioms for union definitions in the ULD meta-data-model.

relationship. For example, assuming we have access to the RDF(S) data-model definition, we can use this new axiom with Axiom 27 to determine that the following query is not well-formed (we discuss these issues in more detail in Chapter 4).

ans(R) $\leftarrow$ d-inst(X, Y), c-inst(X, resource), Y.hasRange:R.

In particular, because we know (from the definition of the data model) that a resource can conform to a class (therefore, Y should be an instance of a class according to our new axiom) and classes do not have range selectors (from Axiom 27), we can infer that the above query is not valid.

2.7 Summary

In this chapter, we defined the ULD and compared it to typical meta-data-model architectures. The ULD offers a flat representation for storing data model constructs, schema, and data. The ULD can also be used to represent data models that permit optional, partial, and multiple levels of schema. We gave a syntax and first-order logic interpretation for representing configurations using the ULD. We introduced the ULD query language and constraint language. The query language is based on stratified Datalog$\neg$ and the constraint language extends the query language to permit existential restrictions. And finally, we used the constraint language to specify a partial axiomatization of the ULD.

In the next chapter, we use the ULD to define a wide variety of data models, including their constructs and constraints. The data models chosen demonstrate the power of the ULD to model various styles of schema conformance. We identify and give a set of short-hand notations for expressing common data-model constraints. And finally, we describe a set of properties that can be used to characterize data models in terms of their conformance requirements.

Chapter 3

Modeling Data Models

In this chapter, we use the ULD to give complete descriptions of a number of existing data models. Our goal is to support our claim that the ULD can represent a wide variety of data models. We consider both constructs and constraints of data models, including conformance definitions. To help specify structural constraints for defining data models, we introduce *constraint macros* in Section 3.1. A constraint macro is a short-hand notation for specifying commonly-used constraints over constructs. Section 3.2 presents ULD representations for the relational [33], E-R [27], and object-oriented [48] data models, which we consider traditional database representations. In Section 3.2, we extend the definition of the relational data model given in Chapter 2 with constraints. Section 3.3 presents ULD representations for the XML [22] and RDF [47] data models, which are primarily used for storing structured information on the Web. We consider XML with DTDs [22], which provide a schema language for XML, and RDF with RDF Schema [23], which serves as the base schema language for RDF, for example, OWL Lite, OWL DL, and OWL Full [56] are layered on top of RDF Schema. We extend the XML and RDF definitions given in Chapter 2 with constraints and additional constructs. In Section 3.4, we present the Topic Map [13] and VIKI [55] hypertext data models. The Topic Map data model is similar to RDF in that it can be used to annotate Web-based resources (such as Web-pages), however it consists of fundamentally different structuring primitives more closely related to structures found in hypertext systems than in semantic networks. The VIKI data model is a structured hypertext data model that is inherently "schema-later" in that tools based on VIKI allow users to create structured, un-typed data, but suggest possible schema types when appropriate. Finally, Section 3.5 introduces a scheme for classifying properties of conformance, which we use to summarize the data models presented in this chapter.

There are a number of ways that constraints can be exploited for data models, and we briefly mention some uses in this chapter (we further discuss uses of constraints in Chapter 4). For a number of data models, constraints can be used not only to clarify and ensure valid configurations, but also to perform basic reasoning services over structures, such as type inference given a schema. In addition, because constraints can provide specification of conformance relationships, they can help further classify data models in terms of their allowed schema-instance arrangements.

3.1 Expressing Common Constraints

We use constraint macros to easily express common structural constraints for data models. Macros are typically more concise for expressing constraints than the constraint language introduced in Chapter 2, but they are not as general: Constraint macros can only be used for certain classes of constraints. (Note that one could define additional forms of macros, not described here, as needed.) We consider macros for expressing uniqueness constraints, which are similar to keys in a relational database, basic collection constraints, and inclusion constraints, which are similar to foreign keys in the relational data model. Each macro described in this section produces a corresponding expression in the constraint language defined in Chapter 2.

3.1.1 Uniqueness Constraints

Given a construct and a set of non-empty component selectors, a *uniqueness constraint* requires each instance of the construct in a configuration to have unique values for the corresponding selectors. The syntax for a uniqueness constraint macro takes the following form, where c is the name (identifier) of a construct, s_1 to s_n are component selectors of c, $n \geq 1$, and t is the name of the constraint.

constraint t: (s_1, s_2, ..., s_n keyOf c).

The expression requires every instance of c to have unique values for s_1 to s_n in a configuration. For example, the following constraint (taken from Chapter 2) expresses a typical uniqueness restriction using the ULD constraint language. The constraint requires all RDF classes in a configuration to have a unique label value.

constraint key1:

C1=C2 $\Leftarrow$ c-inst(C1, class), c-inst(C2, class), C1.hasLabel:L, C2.hasLabel:L.

This constraint can be equivalently expressed using the following uniqueness constraint macro.

constraint key1: (hasLabel keyOf class).

When a uniqueness constraint is expressed against a tuple construct (as in the previous example), the restriction produces the following assertion expressed in the ULD constraint language, where c is a construct, s_1 to s_n are the given component selectors of c, and $n \geq 1$.

X_1=X_2 $\Leftarrow$ c-inst(X_1, c), c-inst(X_2, c),

$\mathrm{X}_1.s_1$:Y_1, $\mathrm{X}_1.s_2$:Y_2, ..., $\mathrm{X}_1.s_n$:Y_n,

$\mathrm{X}_2.s_1$:Y_1, $\mathrm{X}_2.s_2$:Y_2, ..., $\mathrm{X}_2.s_n$:Y_n.

If a uniqueness constraint requires every instance of a tuple construct to be unique in a configuration, we say the constraint is *configuration based*, that is, the constraint limits all construct instances within the configuration. In our previous example with RDF class labels, each instance of the class construct must contain a unique label value in the configuration. Thus, the constraint is configuration-based as are all uniqueness constraints expressed directly against tuple constructs.

Alternatively, a *collection-based* uniqueness constraint restricts tuple instances only within a collection as opposed to within the entire configuration. For a collection construct c_1 defined over a tuple construct c_2, a collection-based uniqueness constraint restricts the members of c_1 to those instances of c_2 that have unique selector values. The macro syntax for defining collection-based uniqueness constraints is identical to the syntax for expressing configuration-based uniqueness constraints. (We use the type of construct being restricted to distinguish between collection-based and configuration-based uniqueness constraints.) For instance, the following collection-based uniqueness constraint requires attribute lists (that is, instances of the `attList` construct) in the relational data model to contain uniquely named `attribute` instances. Thus, `attList` collections may not contain `attribute` items with the same name.

constraint key2: (hasName keyOf attList)

A constraint expressed using a collection-based uniqueness macro produces the following ULD constraint language assertion, where c is the collection construct, s_1 to s_n are the associated component selectors, and $n \geq 1$.

$$\begin{aligned}
&\mathrm{X}_1{=}\mathrm{X}_2 \Leftarrow \text{c-inst}(\mathrm{X}, c),\ \mathrm{X}_1{\in}\mathrm{X},\ \mathrm{X}_2{\in}\mathrm{X},\\
&\quad \mathrm{X}_1.s_1{:}\mathrm{Y}_1,\ \mathrm{X}_1.s_2{:}\mathrm{Y}_2,\ ...,\ \mathrm{X}_1.s_n{:}\mathrm{Y}_n,\\
&\quad \mathrm{X}_2.s_1{:}\mathrm{Y}_1,\ \mathrm{X}_2.s_2{:}\mathrm{Y}_2,\ ...,\ \mathrm{X}_2.s_n{:}\mathrm{Y}_n.
\end{aligned}$$

Certain collection-based uniqueness constraints can be inferred from configuration-based uniqueness constraints. Namely, if a set construct c_1 is defined over c_2 and c_2 has a uniqueness constraint over selectors s_1 to s_n, a collection-based uniqueness constraint also implicitly exists for c_1 over selectors s_1 to s_n. In addition, constraints similar to relational superkeys [71] can be derived from uniqueness restrictions. For a construct with selectors S and a uniqueness constraint over selectors S_k for $\mathrm{S}_k \subseteq \mathrm{S}$, every set of selectors $\mathrm{S}_k \cup \{s\}$ for one or more $s \in \mathrm{S}{-}\mathrm{S}_k$ is also a valid, although more general, uniqueness constraint for the construct.

Finally, we allow configuration-based uniqueness constraints to be defined over union constructs in addition to tuple constructs. For example, consider a union construct c that groups tuple constructs c_1 and c_2, and both c_1 and c_2 share a component selector s. Defining s as a key for c, for example, using the uniqueness constraint (s keyOf c), restricts the set of instances of c_1 and c_2 in a configuration such that each instance has a unique value for s. Note that for the case of union constructs, the same generated ULD constraint (above) still correctly defines uniqueness.

3.1.2 Collection Constraints

Here we describe two additional collection-based constraint macros. The first is called the *non-empty collection* constraint macro, which asserts that every instance of a given collection construct in a configuration contains at least one item. The syntax and corresponding ULD constraint-language assertion for the non-empty collection macro is given below, where c is a collection construct and t is the name of the constraint.

constraint t: (c isNonEmpty)

$\mathrm{Y}\in\mathrm{X} \Leftarrow \text{c-inst}(\mathrm{X}, c).$

The second macro is called the *distinct-collection* constraint and states that every instance of a given collection construct contains unique values. For example, given a list construct c with a distinct-collection constraint, each instance of c in a configuration must contain a unique collection of identifiers. The macro takes the following form, where c is a collection construct and t is the name of the constraint.

constraint t: (c isDistinct)

If c is a set construct, the following ULD constraint-language assertion is generated from the distinct-collection constraint macro.

$\mathrm{X}_1{=}\mathrm{X}_2 \Leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \neg\text{hasDifferentSetItem}(\mathrm{X}_1, \mathrm{X}_2).$

$\text{hasDifferentSetItem}(\mathrm{X}_1, \mathrm{X}_2) \leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \mathrm{Y}\in\mathrm{X}_1, \neg\mathrm{Y}\in\mathrm{X}_2.$
$\text{hasDifferentSetItem}(\mathrm{X}_1, \mathrm{X}_2) \leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \mathrm{Y}\in\mathrm{X}_2, \neg\mathrm{Y}\in\mathrm{X}_1.$

Similarly, if c is a bag construct, the following ULD constraint-language assertion is generated. Recall that the expression $y \in\in x = n$ states that y is in the bag x, n times.

$\mathrm{X}_1{=}\mathrm{X}_2 \Leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \neg\text{hasDifferentBagItem}(\mathrm{X}_1, \mathrm{X}_2).$

$\text{hasDifferentBagItem}(\mathrm{X}_1, \mathrm{X}_2) \leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \mathrm{Y}\in\mathrm{X}_1, \neg\mathrm{Y}\in\mathrm{X}_2.$
$\text{hasDifferentBagItem}(\mathrm{X}_1, \mathrm{X}_2) \leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \mathrm{Y}\in\mathrm{X}_2, \neg\mathrm{Y}\in\mathrm{X}_1.$
$\text{hasDifferentBagItem}(\mathrm{X}_1, \mathrm{X}_2) \leftarrow \text{c-inst}(\mathrm{X}_1, c), \text{c-inst}(\mathrm{X}_2, c), \mathrm{Y}\in\mathrm{X}_1, \mathrm{Y}\in\mathrm{X}_2,$
$\quad \mathrm{Y}\in\in\mathrm{X}_1{=}\mathrm{N}, \mathrm{Y}\in\in\mathrm{X}_2{\neq}\mathrm{N}.$

Finally, for a list construct c, the following ULD constraint-language assertion is generated from the distinct-collection macro.

X_1=X_2 ⇐ c-inst(X_1, c), c-inst(X_2, c), ¬hasDifferentListItem(X_1, X_2).

hasDifferentListItem(X_1, X_2) ← c-inst(X_1, c), c-inst(X_2, c), X_1[I]=Y, X_2[I]≠Y.
hasDifferentListItem(X_1, X_2) ← c-inst(X_1, c), c-inst(X_2, c), X_2[I]=Y, X_1[I]≠Y.

3.1.3 Inclusion Constraints

Inclusion constraint macros are used to ensure that certain construct instances always participate, that is, are always *included*, in some other construct instance. Here we consider macros for membership, joined-membership, joined-subset, and component inclusion constraints.

An example of an inclusion constraint occurs in the relational model, in which every tuple must belong to a table. In other words, it is not possible for a tuple to exist independently of a table. Because tuples are members of table collections, we call the restriction a *membership* inclusion constraint. The macro for a membership inclusion constraint takes the following form, where c_1 is a construct, c_2 is a collection construct defined over c_1, and t is the name of the constraint.

constraint t: (c_1 always(∈) c_2).

The constraint states that every instance of c_1 in a configuration is always a member of an instance of c_2. For example, we would use the macro (tuple always(∈) table) to express the previous relational inclusion constraint. The following ULD constraint-language assertion is produced by a membership inclusion macro.

c-inst(Y, c_2), X∈Y ⇐ c-inst(X, c_1).

Often, it is useful to define inclusion constraints in the context of other constructs. For example, in the relational model, every attribute must be associated with a particular relation. This restriction can be captured using the following inclusion macro, where the `relation` construct has the selector definition `hasAtts` that leads to the `attList` construct, which is defined as a list of attributes.

constraint inc-1: (attribute always($\in$) relation.hasAtts).

The constraint requires that each attribute is a member of at least one relation's attribute list. As shown, we allow inclusion constraints to contain simple path expressions. A simple path expression has the form $c.s_1.s_2...s_n$, where c is a tuple construct (or a union of tuple constructs), and s_1 to s_n are tuple selectors. We call c the *base* construct of the path expression. The following expression represents the general form for membership inclusion macros with simple path expressions, where c_1 and c_2 are base constructs, $m \geq 0$, $n \geq 0$, and t is the name of the construct.

constraint t: ($c_1.s_{11}.s_{12}...s_{1n}$ always($\in$) $c_2.s_{21}.s_{22}...s_{2m}$).

The following ULD constraint-language assertion is produced by the general form of the membership inclusion macro.

c-inst(Y, c_2), Y.s_{21}:Y$_1$, Y$_1$.s_{22}:Y$_2$, ..., Y$_{m-1}$.s_{2m}:Y$_m$, X$_n$$\in$Y$_m$ $\Leftarrow$
c-inst(X, c_1), X.s_{11}:X$_1$, X$_1$.s_{12}:X$_2$, ..., X$_{n-1}$.s_{1n}:X$_n$.

To illustrate, the following ULD constraint is generated from the previous relational macro (inc-1).

c-inst(Y, relation), Y.hasAtts:Y$_1$, X$\in$Y$_1$ $\Leftarrow$ c-inst(X, attribute).

A *joined* inclusion-constraint macro is a special case of an inclusion-constraint macro in which the base construct c_1 is equal to the base construct c_2 (that is, c_1 and c_2 are the same identifier). In a joined inclusion-constraint macro, we treat each side of the constraint as though it denotes the same base-construct instance. For example, for any instance x of construct c with collection instance y such that $x.s$:y, the joined inclusion-constraint macro (c always$\bowtie$($\in$) $c.s$) requires x to be a member of y. (Note that we use always$\bowtie$ to distinguish a joined inclusion-constraint macro from a normal inclusion-constraint macro.)

The following expression represents the general form for joined membership inclusion-constraint macros, where c is the base construct, $m \geq 0$, $n \geq 0$, and t is the name of the constraint.

constraint t: ($c.s_{11}.s_{12}...s_{1n}$ always$\bowtie(\in)$ $c.s_{21}.s_{22}...s_{2m}$).

The following ULD constraint-language assertion is generated for joined membership inclusion-constraint macros, where c denotes the base construct.

$$\begin{aligned}&\text{X}.s_{21}\text{:Y}_1,\ \text{Y}_1.s_{22}\text{:Y}_2,\ ...,\ \text{Y}_{m-1}.s_{2m}\text{:Y}_m,\ \text{X}_n\in\text{Y}_m \Leftarrow\\ &\quad \text{c-inst(X},c\text{)},\ \text{X}.s_{11}\text{:X}_1,\ \text{X}_1.s_{12}\text{:X}_2,\ ...,\ \text{X}_{n-1}.s_{1n}\text{:X}_n.\end{aligned}$$

For example, consider the following joined membership-inclusion constraint for the E-R data model (see Section 3.2.3). A relationship type in the E-R data model contains a list of roles, which can optionally be named. The roleName construct assigns a name to a particular role for a relationship type. The constraint below requires the role being named to be a valid role for the relationship type.

constraint inc-2: (roleName.forRole always$\bowtie(\in)$ roleName.forType.hasRoles).

The following ULD constraint is generated from the macro.

X.forType:Y, Y.hasRoles:Y_1, $\text{X}_1 \in \text{Y}_1 \Leftarrow$ c-inst(X, roleName), X.forRole:X_1.

We consider *subset* and *component* inclusion constraints in addition to membership inclusion constraints. A subset inclusion constraint occurs between two collection constructs[1] and requires a subset relationship between their instances. Here, we only consider joined subset inclusion-constraint macros, which take the following form, where c is the base construct, $n \geq 0$, $m \geq 0$, and t is the name of the constraint. (We discuss general subset inclusion constraints below.)

constraint t: ($c.s_{11}.s_{12}...s_{1n}$ always$\bowtie(\subseteq)$ $c.s_{21}.s_{22}...s_{2m}$).

[1]The subset relation is applied to lists and bags as if they were sets, that is, bags and lists are viewed as sets when testing for a subset relationship.

The following constraint is generated for each joined subset-inclusion macro for base construct c.

> X.s_{21}:Y$_1$, Y$_1$.s_{22}:Y$_2$, ..., Y$_{m-1}$.s_{2m}:Y$_m$, Z$\in$Y$_m$ $\Leftarrow$
> c-inst(X, c), X.s_{11}:X$_1$, X$_1$.s_{12}:X$_2$, ..., X$_{n-1}$.s_{1n}:X$_n$, Z$\in$X$_n$.

The next relational data-model constraint is an example of a joined subset-inclusion macro. A primary-key construct consists of a list of attributes and a relation (which is the relation that the primary key restricts). The constraint below states that the primary-key attributes are a subset of the attributes of the relation that the primary key is defined for.

> constraint inc-3: (pKey.keyAtts always$\bowtie$($\subseteq$) pKey.forRel.hasAtts)

The following ULD constraint is generated from the macro.

> X.forRel:Y$_1$, Y$_1$.hasAtts:Y$_2$, Z$\in$Y$_2$ $\Leftarrow$ c-inst(X, pKey), X.keyAtts:X$_1$, Z$\in$X$_1$.

A component inclusion-constraint macro requires every instance x of a construct c_1 (or an instance of the path started by c_1) to participate as a value of the selector s of some instance y of c_2 (or an instance of the path started by c_2). That is, in the case where no additional paths are given for c_1 or c_2, $y.s{:}x$ must be true. The component inclusion macro takes the following form, where c_1 and c_2 are constructs, $n \geq 0$, $m \geq 0$, and t is the name of the constraint.

> constraint t: ($c_1.s_{11}.s_{12}...s_{1n}$ always(=) $c_2.s_{21}.s_{22}...s_{2m}.s$).

The following ULD constraint-language assertion is produced by a component inclusion macro for base constructs c_1 and c_2.

> c-inst(Y,c_2), Y.s_{21}:Y$_1$, Y$_1$.s_{22}:Y$_2$, ..., Y$_{m-1}$.s_{2m}:Y$_m$, Y$_m$.s:X$_n$ $\Leftarrow$
> c-inst(X,c_1), X.s_{11}:X$_1$, X$_1$.s_{12}:X$_2$, ..., X$_{n-1}$.s_{1n}:X$_n$.

As an example, the following component inclusion macro requires all attList constructs in the relational model to occur in a relation.

constraint inc-4: (attList always(=) relation.hasAtts)

The following ULD constraint is generated from the macro.

c-inst(Y, relation), Y.hasAtts:X $\Leftarrow$ c-inst(X, attList).

We do not consider joined component inclusion-constraint macros because none of the data models described in this chapter require the constraint. However, generating a ULD constraint assertion for a joined component inclusion constraint is straightforward.

Finally, we note the following two properties of inclusion constraint macros. First, for any construct c, the macro ($c.s_1...s_n$ always$\bowtie$($\subseteq$) $c.s_1...s_n$) for $n \geq 0$ is guaranteed to be satisfied in every configuration (assuming the macro is well-formed). Second, certain inclusion constraints are implied by other, more general inclusion constraints, which we list below. We use p and p' to denote arbitrary path expressions. In addition, we use the shorthand notation $p \rightsquigarrow c$ (that is, p "leads to" c) if p is a path expression $c_1.s_1.s_2...s_n$ such that $c_1.s_1$->c_2, c_2.s$_2$->c_3, ..., and $c_{n-1}.s_n$->c, for $n \geq 0$. Note that, if $n = 0$ and $p = c$, then trivially $c \rightsquigarrow c$.

- (c_1 always($\in$) p') implies (p always($\in$) p'), where $p \rightsquigarrow c_1$.
- (c_1 always(=) $p'.s$) implies (p always(=) $p'.s$), where $p \rightsquigarrow c_1$.
- ($c.s_{11}...s_{1n}$ always$\bowtie$($\subseteq$) $c.s_{21}...s_{2m}$) implies ($p'.s_{11}...s_{1n}$ always$\bowtie$($\subseteq$) $p'.s_{21}...s_{2m}$), for all p' such that $p' \rightsquigarrow c$.
- (p always$\bowtie$($\in$) p') implies (p always($\in$) p').

3.1.4 Digression: Regular Subset Inclusion Constraints

We end this section by briefly discussing regular subset inclusion-constraint macros, which cannot be converted to equivalent ULD constraints. We note that for the data models described in this chapter, regular inclusion constraints are not required. However, considering non-joined subset inclusion constraints highlights a possible extension to the ULD constraint language.

A general subset-inclusion macro takes the following form, where c_1 and c_2 are constructs, $n \geq 0$, $m \geq 0$, and t is the name of the constraint.

constraint t: ($c.s_{11}.s_{12}...s_{1n}$ always($\subseteq$) $c.s_{21}.s_{22}...s_{2m}$).

For example, given collection constructs c_1 and c_2, the macro (c_1 always($\subseteq$) c_2) states that for every instance i_1 of c_1 there is an instance i_2 of c_2 such that each member of i_1 is also a member of i_2.

As noted, subset inclusion-constraint macros cannot be expressed in the ULD constraint language, but the previous constraint can be expressed in first-order logic as follows.

$\forall x \exists y \forall z$ (c-inst(x,c_1) $\wedge$ $z \in x \Rightarrow$ c-inst(y,c_2) $\wedge$ $z \in y$)

Notice that this expression is equivalent to the first-order logic expression: $\forall x \exists y \neg \exists z$ (c-inst(x,c_1) $\wedge$ $z \in x$ $\wedge$ $\neg$(c-inst(y,c_2) $\wedge$ $z \in y$)), which can be read as "For every x there is a y that makes the statement 'there is a z in x that is not in y' false." To convert the first-order logic expression to an equivalent ULD constraint-language assertion requires the ability to iterate over (using negation) each z in a collection x for a particular y. However, we believe it is possible to support subset inclusion constraints by extending the ULD constraint language to include intensional predicates in the head of a constraint assertion, as shown in the following program. Note that such an extension does not require any modification to the Datalog¬ representation of ULD constraints presented in Chapter 2.

c-inst(Y,c_1), subset(X,Y) $\Leftarrow$ c-inst(X,c_1).
subset(X,Y) $\leftarrow$ c-inst(X,c_1), c-inst(Y,c_2), ¬notSubset(X,Y).
notSubset(X,Y) $\leftarrow$ c-inst(X,c_1), c-inst(Y,c_2), Z∈X, ¬Z∈Y.

3.2 Traditional Database Data Models

This section gives ULD descriptions for traditional database data models. We present the relational data model in Section 3.2.1, with identical constructs as those in Example 1 of Chapter 2, but extended with constraints. We then consider a typical object-oriented data model in Section 3.2.2. And finally, we describe the Entity-Relationship (E-R) data model [27] in Section 3.2.3, which combines some of the structures of the relational and object-oriented data models, but with slightly looser conformance constraints.

3.2.1 Relational Data Models

The relational data model was first introduced by Codd [33] and is the most widely used and implemented data model for databases [71]. The relational data-model constructs and constraints are given below (Data-Model 1). Note that we assume that a configuration stores information for a single relational database and that there is a special, atomic value *null* that is designated to represent the null value.

In the relational-model specification given below, the relation construct corresponds to a table definition, where each relation in a configuration has a unique name and a list of attribute definitions. A table serves as the extent for a relation. Each table must conform to exactly one relation, and a relation has exactly one table as an instance. Thus, tables and relations are one-to-one. Tables consist of tuples, and each tuple in a table contains a value for each attribute of the table's corresponding relation. We model a tuple as a list of values, where the i-th value is a value for the i-th attribute of the relation. The number of values in a tuple is precisely the number of attributes in its corresponding relation.

We consider the typical integrity constraints for the relation model, that is, primary and foreign keys. Each relation can have at most one primary key (`pKey`). A primary key designates one or more relation attributes as unique, that is, the combination of values that form the primary key must be unique for each tuple in a table. A foreign key (`fkey`) defines a referential integrity constraint in which each tuple in a table can reference a key-value in a target table. A foreign key constraint requires the designated values of the foreign-key attributes in a source relation to contain either corresponding values of the primary-key attributes in a target relation or *null* values.

As shown, rel-1 through rel-16 are constraints for the relational data model that can be expressed using macros. The first four constraints state that attribute lists, primary- and foreign-key attribute lists, and tuples are non-empty collections. Thus, each relation must have at least one attribute, and similarly, each tuple must have at least one value.

If a primary or foreign key is defined, it must contain at least one attribute. The next six constraints, rel-5 through rel-10, define uniqueness restrictions. In particular, every relation in a configuration has a unique name, each attribute in an attribute list (including those for foreign and primary keys) has a unique name, each relation can have at most one primary key (that is, each primary key must have a unique forRel value), and each relation can have at most one foreign key for a particular set of attributes and target relation. Constraint rel-11 requires attributes to be nested within attribute lists (that is, every attribute must be assigned to some relation). Constraints rel-12 and rel-13 require primary- and foreign-key attribute lists to always be associated with a primary and foreign key, respectively. Constraint rel-14 requires tuples to always be contained within a table. The last two constraints (rel-15 and rel-16) define the subset relationship between a primary (foreign) key's attribute list and the corresponding relation's attribute list.

The next three constraints, rel-17 through rel-19, cannot be expressed using macros. Each of these constraints further defines the foreign-key construct. The first constraint (rel-22) requires each foreign key to reference (via the toRel selector) a relation with a primary key. The second constraint (rel-23) requires the number of attributes in the source relation designated by the foreign key to be the same as the number of attributes in the target relation's primary key. The last constraint (rel-24) requires the source attributes to have the same domain as their corresponding attributes in the target relation's primary key. We note that the attributes of the source relation designated by the foreign key are assumed to be ordered to match the order of the primary key attributes in the target relation.

The remaining constraints define conformance for the relational model. The first three conformance constraints define the relationship between tables, tuples, relations, and attributes. The last two constraints describe primary- and foreign-key restrictions on tuples. To model the foreign-key restriction, we use a constraint assertion that is not satisfiable; a positive *d-inst* formula appears in the body of the rule and a negative *d-inst* formula appears in the head of the rule, where both formulas have identical parameters. Note that the body of the assertion succeeds only if the foreign-key restriction is violated. Thus, whenever the body of the constraint holds the constraint assertion derives (or requires) a contradiction. Therefore, although the assertion can never be satisfied, it accurately captures the semantics of the foreign-key restriction.

Data-Model 1 (*The relational data-model constructs and constraints*)

```
construct relation    = {hasName->uldString, hasAtts->attList}
construct attList     = list of attribute
construct attribute   = {hasName->uldString, hasDomain->uldValuetype}
construct pKey        = {forRel->relation, keyAtts->pKeyAttList}
construct fKey        = {forRel->relation, toRel->relation, keyAtts->fKeyAttList}
construct pKeyAttList = list of attribute
construct fKeyAttList = list of attribute
construct table       = bag of tuple conf(domain=1,range=1):relation
construct tuple       = list of uldValue conf(domain=*,range=1):relation
```

constraint rel-1: (attList isNonEmpty).
constraint rel-2: (pKeyAttList isNonEmpty).
constraint rel-3: (fKeyAttList isNonEmpty).
constraint rel-4: (tuple isNonEmpty).

constraint rel-5: (hasName keyOf relation).
constraint rel-6: (hasName keyOf attList).
constraint rel-7: (forRel keyOf pKey).
constraint rel-8: (forRel, toRel, keyAtts keyOf fKey).
constraint rel-9: (hasName keyOf pKeyAttList).
constraint rel-10: (hasName keyOf fKeyAttList).

constraint rel-11: (attList always(=) relation.hasAtts).
constraint rel-12: (attribute always(∈) attList).
constraint rel-13: (pKeyAttList always(=) pKey.keyAtts).
constraint rel-14: (fKeyAttList always(=) fKey.keyAtts).
constraint rel-15: (pKey.keyAtts always⋈(⊆) pKey.forRel.hasAtts).
constraint rel-16: (fKey.keyAtts always⋈(⊆) fKey.forRel.hasAtts).

constraint rel-17: (*A foreign key always refers to a target relation that has a primary key.*)
c-inst(P, pKey), P.forRel:R $\Leftarrow$ c-inst(F, fKey), F.toRel:R.

constraint rel-18: (*Number of foreign key attributes must equal number of key attributes.*)
$|A_p|$=N $\Leftarrow$ c-inst(F, fKey), F.toRel:R, c-inst(P, pKey), P.forRel:R, P.keyAtts:A_p,

F.keyAtts:A_f, $|A_f|$=N.

constraint rel-19: (*Domain of foreign key attributes must equal domain of key attributes.*)
A_p[I]=X_2, X_2.hasDomain:D $\Leftarrow$ c-inst(F, fKey), F.toRel:R, c-inst(P, pKey), P.forRel:R, P.keyAtts:A_p, F.keyAtts:A_f, A_f[I]=X_1, X_1.hasDomain:D.

constraint rel-conf1: (*Every tuple in a table conforms to the corresponding relation*)
d-inst(T, R) $\Leftarrow$ c-inst(B, table), c-inst(R, relation), d-inst(B, R), T$\in$B.

constraint rel-conf2: (*Tuple values conform to corresponding attribute domains*)
d-inst(V, D) $\Leftarrow$ c-inst(T, tuple), c-inst(R, relation), d-inst(T, R), R.hasAtts:A_R, A_r[I]=A, A.hasDomain:D, T[I]=V.

constraint rel-conf3: (*Tuples have the same number of values as relation attributes*)
|T|=N $\Leftarrow$ c-inst(T, tuple), c-inst(R, relation), d-inst(T, R), R.hasAtts:A_R, $|A_r|$=N.

constraint rel-conf4: (*Primary key constraint*)
T_1=T_2 $\Leftarrow$ c-inst(T_1, tuple), c-inst(T_2, tuple), d-inst(T_1, R), d-inst(T_2, R), ¬differ(T_1, T_2, R).

differ(T_1, T_2, R) $\leftarrow$ c-inst(T_1, tuple), c-inst(T_2, tuple), d-inst(T_1, R), d-inst(T_2, R), c-inst(P, pKey), P.forRel:R, P.keyAtts:A_p, R.hasAtts:A_r, $X_1$$\in$$A_p$, $X_2$$\in$$A_r$, X_1.hasName:N, X_2.hasName:N, A_r[I]=X_2, T_1[I]=V_1, T_2[I]=V_2, $V_1$$\neq$$V_2$.

constraint rel-conf5: (*Foreign key constraint*)
¬d-inst(T, R_s) $\Leftarrow$ c-inst(F, fKey), F.forRel:R_s, c-inst(T, tuple), d-inst(T, R_s), ¬satisfyFKeyConstraint(T, F).

satisfyFKeyConstraint(T, F) $\leftarrow$ c-inst(F, fKey), F.forRel:R_s, c-inst(T, tuple), d-inst(T, R_s), ¬hasNonNullFKeyAtts(T, F).

satisfyFKeyConstraint(T, F) $\leftarrow$ c-inst(F, fKey), F.forRel:R_s, c-inst(T, tuple), d-inst(T, R_s), ¬hasNonMatchingTuple(T, F).

hasNonNullFKeyAtts(T, F) $\leftarrow$ c-inst(F, fKey), F.forRel:R_s, d-inst(T, R_s), F.keyAtts:AL_f, R_s.hasAtts:AL, AL[I]=A, $A_f$$\in$$AL_f$, A.hasName:N, A_f.hasName:N, T[I]$\neq$null.

hasNonMatchingTuple(T, F) $\leftarrow$ c-inst(F, fKey), F.forRel:R_s, c-inst(T, tuple), d-inst(T, R_s), F.keyAtts:AL_f, R_s.hasAtts:AL, A$\in$AL, AL_f[I]=A_f, A.hasName:N, A_f.hasName:N, ¬matchingAttributeVal(T, I, F).

matchingAttributeVal(T_s, I, F) $\leftarrow$ c-inst(F, fKey), c-inst(P, pKey), F.forRel:R_s, F.toRel:R_t, P.forRel:R_t, c-inst(T_s, tuple), d-inst(T, R_s), c-inst(T_t, tuple), d-inst(T, R_t), R_s.hasAtts:AL_s, R_t.hasAtts:AL_t, $AL_s[I_s]=A_s$, $AL_t[I_t]=A_t$, F.keyAtts:AL_f, $AL_f[I]=A_f$, A_f.hasName:N_f, A_s.hasName:N_f, P.keyAtts:AL_p, $AL_p[I]=A_p$, A_p.hasName:N_p, A_t.hasName:N_p, $T_s[I_s]=V$, $T_t[I_t]=V$.

3.2.2 Object-Oriented Data Models

Object-oriented database data models extend relational data models by adding structured data types and object identifiers (to distinguish among objects with equivalent values) [48]. We present a version of the O_2 data model [48, 49] (Data-Model 2, below), which has played a significant role in shaping the ODMG Object Model [26]—an evolving standard for object-based data management.

We simplify the data model of O_2 as follows. Class names denote implicit extents, that is, every class name can be used as a collection that contains its designated instances. Each class has a tuple-structured type that defines a set of attributes, each of which can be basic (that is, contain atomic types) or structured. A structured type is either another class or a set type, where a set type can be arbitrarily nested with structured types. (In the ODMG Object Model and in O_2, structured types include tuple structures in addition to class and set types). Class types imply an implicit subsumption hierarchy, which may be more complete than the user-specified class hierarchy (that is, there may be classes whose types are compatible, but are not designated as subclasses). We assume each object has an identifier and a value, which is either atomic or structured. We do not consider method signatures, although they can easily be added to the definition below.

In an O_2 database, every object must belong to (that is, be an instance of) at least one class extent. An object is allowed to exist in more than one class extent as allowed by type subsumption. Finally, the O_2 data model allows only single inheritance, although it can be extended to support multiple class inheritance [49].

We note that in both the ODMG Object Model and the O_2 data model, attributes can contain arbitrarily nested tuple structures in addition to the structured types previously defined. However, when arbitrarily nested structures are considered, the corresponding restriction for checking type subsumption (oodm-19 below) cannot be extended using the

ULD constraint language. That is, we cannot construct a stratified Datalog¬ program that accurately captures the class-superclass typing constraint for arbitrarily nested structures (as currently defined). In particular, an additional *compatibleAttribute* rule is needed to ensure that an attribute (variable A_2 in *compatibleAttribute*) with a nested tuple type is present in the given tuple type (variable T_1 in *compatibleAttribute*). Performing this check requires negation through the *incompatibleTypes* formula. Thus, *incompatibleTypes* would call a *compatibleAttribute* formula, and *compatibleAttribute* would call a negated *incompatibleTypes* formula. We note that, from a modeling perspective, little is gained by using nested tuple types, because classes, which can essentially represent named tuple types, can be arbitrarily nested.

Data-Model 2 (*The O_2 data-model constructs and constraints*)

```
construct class       = {hasName->uldString, hasType->tupleType, hasSuper->class}
construct tupleType   = set of attribute
construct attribute   = {hasName->uldString, hasType->nestedType}
construct nestedType  = class | basicType | setType
construct basicType   = {hasType->uldValueType}
construct setType     = {ofType->nestedType}
construct object      = {hasId->uldString, hasValue->tupleValue}
                        conf(domain=*,range=+):class
construct tupleValue  = set of property
construct property    = {hasName->uldString, hasValue->nestedValue}
construct nestedValue = object | basicValue | setValue
construct basicValue  = {hasValue->uldValue}
construct setValue    = set of nestedValue
```

constraint oodm-1: (tupleType isDistinct).
constraint oodm-2: (setType isDistinct).
constraint oodm-3: (tupleValue isDistinct).
constraint oodm-4: (setValue isDistinct).

constraint oodm-5: (hasName keyOf class).
constraint oodm-6: (hasName keyOf tupleType).
constraint oodm-7: (hasId keyOf object).
constraint oodm-8: (hasValue keyOf basicValue).

constraint oodm-9: (hasName keyOf tupleValue).

constraint oodm-10: (attribute always(∈) tupleType).
constraint oodm-11: (property always(∈) tupleValue).

constraint oodm-12: (*Each tuple type must be connected to a class*)
c-inst(C, class), C.hasType:T $\Leftarrow$ c-inst(T, tupleType).

constraint oodm-13: (*Each non-class nested type must be connected to a class*)
c-inst(C, class), C.hasType:T $\Leftarrow$ c-inst(T, nestedType), $\neg$ c-inst(T, class),
$\neg$ hasParentType(T).

hasParentType(T) $\leftarrow$ c-inst(T_s, setType), T_s.ofType:T.
hasParentType(T) $\leftarrow$ c-inst(T_a, attribute), T_a.hasType:T.

constraint oodm-14: (*Each tuple value must be connected to an object*)
c-inst(O, object), O.hasValue:T $\Leftarrow$ c-inst(T, tupleValue)

constraint oodm-15: (*Each non-object nested value must be connected to an object*)
c-inst(O, object), O.hasValue:V $\Leftarrow$ c-inst(V, nestedValue), $\neg$ c-inst(V, object),
$\neg$ hasParentValue(V).

hasParentValue(V) $\leftarrow$ c-inst(V_s, setValue), $V \in V_s$.
hasParentValue(V) $\leftarrow$ c-inst(V_p, property), V_p.hasValue:V.

constraint oodm-16: (*Subclass hierarchy is acyclic*)
$\neg$ C_1.hasSuper:C_2 $\Leftarrow$ c-inst(C_1, class), c-inst(C_2, class), subClassClosure(C_2, C_1).

subClassClosure(C_1, C_1) $\leftarrow$ c-inst(C_1, class).
subClassClosure(C_1, C_3) $\leftarrow$ c-inst(C_1, class), c-inst(C_3, class), C_1.hasSuper:C_2,
subClassClosure(C_2, C_3).

constraint oodm-17: (*Set types cannot be infinite*)
$T_1 \neq T_2$ $\Leftarrow$ hasNestedSetType(T_1, T_2).

hasNestedSetType(T_1, T_2) $\leftarrow$ c-inst(T_1, setType), c-inst(T_2, setType), T_1.ofType:T_2.
hasNestedSetType(T_1, T_3) $\leftarrow$ c-inst(T_1, setType), c-inst(T_3, setType), T_1.ofType:T_2,
hasNestedSetType(T_2, T_3).

constraint oodm-18: (*Set values cannot be recursive*)
$V_1 \neq V_2$ $\Leftarrow$ hasNestedSetValue(V_1, V_2).

hasNestedSetValue(V_1, V_2) $\leftarrow$ c-inst(V_1, setValue), c-inst(V_2, setValue), $V_2 \in V_1$.
hasNestedSetValue(V_1, V_3) $\leftarrow$ c-inst(V_1, setValue), c-inst(V_3, setValue), $V_2 \in V_1$,
hasNestedSetValue(V_2, V_3).

constraint oodm-19: (*Subclass type must be compatible with superclass type.*)
$\neg$ C_1.hasSuper:C_2 $\Leftarrow$ c-inst(C_1, class), c-inst(C_2, class), C_1.hasType:T_1, C_2.hasType:T_2,
incompatibleTypes(T_1, T_2).

incompatibleTypes(T_1, T_2) $\leftarrow$ c-inst(T_1, tupleType), c-inst(T_2, tupleType), $A_2 \in T_2$,
$\neg$ compatibleAttribute(A_2, T_1).

compatibleAttribute(A_2, T_1) $\leftarrow$ $A_1 \in T_1$, A_1.hasName:N, A_2.hasName:N, A_1.hasType:T'_1,
A_2.hasType:T'_2, c-inst(T'_1, class), c-inst(T'_2, class), subClassClosure(T'_1, T'_2).
compatibleAttribute(A_2, T_1) $\leftarrow$ $A_1 \in T_1$, A_1.hasName:N, A_2.hasName:N, A_1.hasType:T'_1,
A_2.hasType:T'_2, c-inst(T'_1, basicType), c-inst(T'_2, basicType), T'_1.hasType:T,
T'_2.hasType:T.
compatibleAttribute(A_2, T_1) $\leftarrow$ $A_1 \in T_1$, A_1.hasName:N, A_2.hasName:N, A_1.hasType:T'_1,
A_2.hasType:T'_2, c-inst(T'_1, setType), c-inst(T'_2, setType), setSubType(T'_1, T'_2).

setSubType(T_1, T_2) $\leftarrow$ c-inst(T_1, setType), c-inst(T_2, setType), T_1.ofType:T'_1,
T_2.ofType:T'_2, c-inst(T'_1, class), c-inst(T'_2, class), subClassClosure(T'_1, T'_2).
setSubType(T_1, T_2) $\leftarrow$ c-inst(T_1, setType), c-inst(T_2, setType), T_1.ofType:T'_1,
T_2.ofType:T'_2, c-inst(T'_1, basicType), c-inst(T'_2, basicType), T'_1.hasType:T,
T'_2.hasType:T.
setSubType(T_1, T_2) $\leftarrow$ c-inst(T_1, setType), c-inst(T_2, setType), T_1.ofType:T'_1,
T_2.ofType:T'_2, c-inst(T'_1, setType), c-inst(T'_2, setType), setSubType(T'_1, T'_2).

constraint oodm-conf1: (*Objects are instances of their corresponding superclasses*)
d-inst(O, C_2) $\Leftarrow$ c-inst(O, object), d-inst(O, C_1), C_1.hasSuper:C_2.

constraint oodm-conf2: (*Objects must have compatible types to be class instances*)
$\neg$d-inst(O, C) $\Leftarrow$ c-inst(O, object), c-inst(C, class), O.hasValue:V, C.hasType:T,
incompatibleValueForType(V, T).

incompatibleValueForType(V, T) $\leftarrow$ c-inst(V, tupleValue), c-inst(T, tupleType), $A \in T$,
$\neg$ hasPropertyForAttribute(V, A).

hasPropertyForAttribute(V, A) $\leftarrow$ c-inst(V, tupleValue), c-inst(A, attribute),
V.hasName:N, A.hasName:N, V.hasValue:V', A.hasType:T, c-inst(V', object),
c-inst(T, class), d-inst(V', T).
hasPropertyForAttribute(V, A) $\leftarrow$ c-inst(V, tupleValue), c-inst(A, attribute),

V.hasName:N, A.hasName:N, V.hasValue:V′, A.hasType:T,
c-inst(V′, basicValue), c-inst(T, basicType), V′.hasValue:X, T.hasType:T′,
d-inst(X, T′).

hasPropertyForAttribute(V, A) ← c-inst(V, tupleValue), c-inst(A, attribute),
V.hasName:N, A.hasName:N, V.hasValue:V′, A.hasType:T, c-inst(V′, setValue),
c-inst(T, setType), ¬ incompatValueForSetType(V′, T).

incompatibleValueForSetType(V, T) ← c-inst(V, setValue), c-inst(T, setType), X∈V,
T.hasType:T′, c-inst(X′, setValue), ¬ c-inst(T′, setType).

incompatibleValueForSetType(V, T) ← c-inst(V, setValue), c-inst(T, setType), X∈V,
T.hasType:T′, c-inst(X, basicValue), ¬ c-inst(T′, basicType).

incompatibleValueForSetType(V, T) ← c-inst(V, setValue), c-inst(T, setType), X∈V,
T.hasType:T′, c-inst(X, objectValue), ¬ c-inst(T′, class).

incompatibleValueForSetType(V, T) ← c-inst(V, setValue), c-inst(T, setType), X∈V,
c-inst(X, basicValue), X.hasValue:X′, T.hasType:T′, c-inst(T′, basicType),
¬ d-inst(X′, T′).

incompatibleValueForSetType(V, T) ← c-inst(V, setValue), c-inst(T, setType), X∈V,
c-inst(X, object), T.hasType:T′, c-inst(T′, class), ¬ d-inst(X, T′).

incompatibleValueForSetType(V, T) ← c-inst(V, setValue), c-inst(T, setType), X∈V,
c-inst(X, setValue), T.hasType:T′, incompatibleValueForSetType(X, T′).

3.2.3 Entity-Relationship Data Models

The E-R data model [27] is primarily used as a conceptual modeling language for designing database schemas [71]. In particular, it can be used to define a conceptual schema that can then be mapped to relational table definitions (the logical schema) for storing data.

Information in the E-R data model is separated into entities and (explicit) relationships among entities. Both entities and relationships are grouped into types forming an E-R schema. An entity type consists of one or more attribute definitions. Attributes within an entity type can form keys, and each entity type must have exactly one primary key. A relationship type, which links two or more entity types, may also have attributes (although they are not required). Relationships do not have explicit keys. Instead, the primary key for a relationship is the combination of the primary keys of the entities being related. A relationship type, however, can be constrained using cardinality restrictions, which limit the participation of entities in the instances of the relationship type. A relationship type may also define role names for each entity type that participates in the relationship. The role name represents the function an entity plays in the relationship and is useful for distinguishing the roles played in a recursive relationship type, that is, a relationship type that connects an entity type to itself.

An entity type can be designated as a *weak* entity type when its identity depends on another entity type. A weak entity type is specified using a combination of one or more of its attributes, called a *partial* key, and an *identifying* relationship type that connects the entity type to the entity type it depends on. Note that a partial key is required for a weak entity type. Thus, the primary key of a weak entity type consists of its partial key combined with its identifying entity type's primary key. A typical example of a weak entity is an employee dependent, where the dependent is covered by the employee's health insurance. Little information is recorded for each dependent; for example, a name and a birthdate may be the only attributes. The name of the dependent serves as its *partial* key and a relationship such as *dependent_of* serves as its identifying relationship. A dependent entity is uniquely identified by specifying its name (the partial key) combined with an employee number (assuming the employee number is the primary key for an employee), where the dependent is connected to the employee through a *dependent_of* relationship. Note that for each identifying entity-type instance, the instance's corresponding weak-entity instances are required to have distinct values for the partial key.

In the E-R data model, an entity is considered an instance of an entity type whenever it conforms to the specification of the type. Thus, the instance-of relationship in the E-R data model follows from the structure of entities, which is in contrast to object-oriented data models, where the user must explicitly specify the type of an object. It is also possible for an entity or relationship to exist without being classified by an available entity or relationship type, respectively. Most current applications of the E-R data model, however, assume schema is required, and only consider entities and relationships that are associated with types. This assumption is found in applications that use an E-R schema as a conceptual model for an underlying relational data-model representation.

We describe the E-R data model below (Data-Model 3). We assume that attributes are always single-valued (and not composite), that role names on relationship types are optional, and that cardinality constraints restrict the participation of an entity in a relationship. In addition, we choose the general approach of not requiring entities and relationships to have corresponding types.

Note that a role is created for each actual role of a relationship type. Thus, for a relationship that contains two distinct roles for the same entity type, two instances of the role construct are created and stored for the role list. Therefore, role names and cardinality constraints can identify a particular role in a relationship type by giving a role identifier and a relationship-type identifier. We do not require roles to be unique across relationship types, that is, the same role can appear in multiple role lists. An entity type can also play the same role in multiple relationship types.

Data-Model 3 (*The E-R data-model constructs and constraints*)

```
construct entityType = {hasName->uldString, hasAtts->attSet}
construct attSet     = set of attribute
construct attribute  = {hasName->uldString, hasDomain->uldValueType}
construct relType    = {hasName->uldString, hasRoles->roleList, hasAtts->attSet}
construct roleList   = list of role
construct role       = {hasType->entityType}
construct roleName   = {forType->relType, forRole->role, hasName->uldString}
construct pKey       = {forType->entityType, keyAtts->pKeyAttSet}
construct pKeyAttSet = set of attribute
construct weakEntity = {forType->entityType, idRel->relType,
                       partialKey->weakAttSet}
construct weakAttSet = set of attribute
construct card       = {forType->relType, forRole->role, isReq->uldBool,
                       isMult->uldBool}
construct instance   = entity | rel
construct entity     = {hasProps->propSet} conf(domain=*,range=*):entityType
construct propSet    = set of property
construct property   = {hasName->uldString, hasVal->uldValue}
construct rel        = {hasProps->propSet, associates->entityList}
                       conf(domain=*,range=*):relType
construct entityList = list of entity
```

constraint er-1: (roleList isNonEmpty).
constraint er-2: (entityList isNonEmpty).

constraint er-3: (attSet isDistinct).
constraint er-4: (roleList isDistinct).
constraint er-5: (pKeyAttSet isDistinct).
constraint er-6: (weakAttSet isDistinct).
constraint er-7: (propSet isDistinct).
constraint er-8: (entityList isDistinct).

constraint er-9: (hasName keyOf entityType).
constraint er-10: (hasName keyOf attSet).
constraint er-11: (hasName keyOf relType).
constraint er-12: (forRel, forRole keyOf roleName).
constraint er-13: (forType keyOf pKey).
constraint er-14: (hasName keyOf pKeyAttSet).
constraint er-15: (forType keyOf weakEntity).
constraint er-16: (hasName keyOf weakAttSet).
constraint er-17: (forRel, forRole keyOf card).
constraint er-18: (hasName keyOf propSet).

constraint er-19: (attSet always(=) type.hasAtts).
constraint er-20: (roleList always(=) relType.hasRoles).
constraint er-21: (pKeyAttSet always(=) pKey.keyAtts).
constraint er-22: (weakAttSet always(=) weakEntity.partialKey).
constraint er-23: (propSet always(=) instance.hasProps).
constraint er-24: (entityList always(=) rel.associates).
constraint er-25: (attribute always($\in$) attSet).
constraint er-26: (role always($\in$) roleList).
constraint er-27: (property always($\in$) propSet).
constraint er-28: (pKey.keyAtts always$\bowtie$($\subseteq$) pKey.forType.hasAtts).
constraint er-29: (weakEntity.partialKey always$\bowtie$($\subseteq$) weakEntity.forType.hasAtts).
constraint er-30: (roleName.forRole always$\bowtie$($\in$) roleName.forType.hasRoles).
constraint er-31: (card.forRole always$\bowtie$($\in$) card.forType.hasRoles).

constraint er-32: (*Relationship types have unique sets of role names*)
$R_{n1}=R_{n2} \Leftarrow$ c-inst(R_{n1}, roleName), c-inst(R_{n2}, roleName), R_{n1}.forType:R, R_{n2}.forType:R, R_{n1}.hasName:N, R_{n2}.hasName:N.

constraint er-33: (*Role lists have at least two roles*)
$R_1 \in R_l$, $R_2 \in R_l$, $R_1 \neq R_2 \Leftarrow$ c-inst(R_l, roleList).

constraint er-34: (*A role list contains unique role ids*)

$R_1 \neq R_2 \Leftarrow$ c-inst(R_l, roleList), $R_1 \in R_l$, $R_2 \in R_l$.

constraint er-35: (*Every entity list has at least two elements*)

$E_1 \in E_l$, $E_2 \in E_l$, $E_1 \neq E_2 \Leftarrow$ c-inst(E_l, entityList).

constraint er-36: (*Entity types have at least one attribute*)

$A \in A_s \Leftarrow$ c-inst(E, entityType), E.hasAtts:A_s.

constraint er-37: (*Entities have at least one property*)

$P \in P_l \Leftarrow$ c-inst(E, entity), E.hasProps:P_l.

constraint er-38: (*Weak entity types do not have primary keys*)

$\neg$P.forType:T $\Leftarrow$ c-inst(T, entityType), c-inst(W, weakEntity), W.forType:T,
c-inst(P, pKey).

constraint er-39: (*Non-weak entity types always have primary keys*)

c-inst(P, pKey), P.keyFor:T $\Leftarrow$ c-inst(T, entityType), $\neg$isWeakEntityType(T).

isWeakEntityType(T) $\leftarrow$ c-inst(T, entityType), c-inst(W, weakEntity), W.forType:T.

constraint er-40: (*Identifying relationships are binary and connect to weak entity types*)

$|R_l|=2$, $L_1 \in R_l$, L_1.hasType:E $\Leftarrow$ c-inst(W, weakEntity), W.forType:E, W.idRel:R,
R.hasRoles:R_l.

constraint er-conf1: (*Entities have all attributes of their entity types*)

$P \in P_l$, P.hasName:N, P.hasValue:V, d-inst(V, D) $\Leftarrow$ c-inst(T, entityType), c-inst(E, entity),
T.hasAtts:A_l, $A \in A_l$, A.hasName:N, A.hasDomain:D, E.hasProps:P_l.

constraint er-conf2: (*Relationships have all attributes of their relationship types*)

$P \in P_l$, P.hasName:N, P.hasValue:V, d-inst(V, D) $\Leftarrow$ c-inst(T, relType), d-inst(R, T)
T.hasAtts:A_l, $A \in A_l$, A.hasName:N, A.hasDomain:D, R.hasProps:P_l.

constraint er-conf3: (*Relationships connect a valid entity for each relationship-type role*)

E_l[I]=E, d-inst(E, T_e) $\Leftarrow$ c-inst(T_r, relType), d-inst(R, T), T_r.hasRoles:O_l, O_l[I]=O,
O.hasType:T_e, R.associates:E_l.

constraint er-conf4: (*Primary key constraint*)

$E_1 = E_2 \Leftarrow$ c-inst(E_1, entity), c-inst(E_2, entity), d-inst(E_1, T), d-inst(E_2, T), c-inst(pKey, P), P.forType:T, P.keyAtts:A_k, ¬differOnAttribute(E_1, E_2, A_k).

differOnAttribute(E_1, E_2, A_k) $\leftarrow$ E_1.hasProps:P_{s1}, E_2.hasProps:P_{s2}, $A \in A_k$, $P_1 \in P_{s1}$, $P_2 \in P_{s2}$, A.hasName:N, P_1.hasName:N, P_2.hasName:N, P_1.hasVal:V_1, P_2.hasVal:V_2, $V_1 \neq V_2$.

constraint er-conf5: (*Relationship types have unique instances*)

$R_1 = R_2$ $\Leftarrow$ c-inst(R_1, rel), c-inst(R_2, rel), d-inst(R_1, T), d-inst(R_2, T), T.hasRoles:R_l, ¬differOnRole(R_1, R_2, R_l).

¬differOnRole(R_1, R_2, R_l) $\leftarrow$ c-inst(R_l, roleList), c-inst(R_1, rel), c-inst(R_2, rel), R_l[I]=O_l, R_1.associates:E_{l1}, R_2.associates:E_{l2}, E_{l1}[I]=E_1, E_{l2}[I]=E_2, $E_1 \neq E_2$.

constraint er-conf6: (*Weak entities must have an identifying relationship*)

d-inst(R, T_r), R.associates:E_l, E_l[I]=E $\Leftarrow$ c-inst(W, weakEntity), W.forType:T_e, d-inst(E, T_e), W.idRel:T_r, T_r.hasRoles:R_l, R_l[I]=L, L.hasType:T_e.

constraint er-conf7: (*Partial keys constraint*)

$E_1 = E_2$ $\Leftarrow$ c-inst(W, weakEntity), W.forType:T_{e1}, W.idRel:T_r, d-inst(E_1, T_{e1}), d-inst(E_2, T_{e1}), T_r.hasRoles:R_l, R_l[I_1]=L_1, L_1.hasType:T_{e1}, R_l[I_2]=L_2, L_2.hasType:T_{e2}, $T_{e1} \neq T_{e2}$, d-inst(R_1, T_r), d-inst(R_2, T_r), R_1.associates:E_{l1}, E_{l1}[I_1]=E_1, R_2.associates:E_{l2}, E_{l2}[I_1]=E_2, E_{l1}[I_2]=E, E_{l2}[I_2]=E, ¬differOnPartialAttributes(E_1, E_2, W).

differOnPartialAttributes(E_1, E_2,W) $\leftarrow$ c-inst(W, weakEntity), c-inst(E_1, entity), c-inst(E_2, entity), W.partialKey:A_s, $A \in A_s$, E_1.hasProps:P_{s1}, $P_1 \in P_{s1}$, E_2.hasProps:P_{s2}, $P_2 \in P_{s2}$, A.hasName:N, P_1.hasName:N,P_2.hasName:N, P_1.hasVal:V_1, P_2.hasVal:V_2, $V_1 \neq V_2$.

constraint er-conf8: (*Required cardinality constraint*)

d-inst(R, T_r), R.associates:E_l, E_l[I]=E $\Leftarrow$ c-inst(T_r, relType), T_r.hasRoles:R_l, R_l[I]=L, c-inst(C, card), C.forRole:L, C.forType:T_r, C.isReq:true, L.hasType:T_e, c-inst(E, T_e).

constraint er-conf9: (*Maximum cardinality constraint*)

$R_1 = R_2$ $\Leftarrow$ c-inst(T_r, relType), T_r.hasRoles:R_l, R_l[I]=L, c-inst(C, card), C.forRole:L, C.forType:T_r, C.isMult:false, L.hasType:T_e, c-inst(E, T_e), c-inst(R_1, T_r), c-inst(R_2, T_r), R_1.associates:EL_1, R_2.associates:EL_2, EL_1[I]=E, EL_2[I]=E.

3.3 Structured Data Models for the Web

In this section, we describe data models for XML and RDF, two standards for exchanging information over the Web. In Section 3.3.1, we extend the definition of the XML data model from Chapter 2 by including a full description of the Document Type Definition (DTD) schema language. In Section 3.3.2, we describe the full RDF data model with RDF Schema and briefly mention the flexibility of RDF for defining metamodels by allowing multiple levels of schema.

3.3.1 XML Data Models

The Extensible Markup Language (XML) [22] is a text format for representing and exchanging hierarchically structured information. Data in XML consists of elements, attributes, and character-string data. The data model of XML is hierarchical. An XML document is structured as a tree, where an internal node in the tree is an element, a leaf node in the tree is either an element or string content (called 'PCDATA'), and the children of a node are ordered. An XML document must have exactly one root element. Every element has a tag name and can contain an optional set of attribute-value pairs.

We give a ULD description for the XML data model below, including a description of DTDs for defining schemas for XML documents. As shown, the majority of constructs introduced in the ULD description below are for representing the structure of DTDs—all but the last ten constructs describe the structure of DTDs.

With a DTD, it is possible to restrict the allowable elements and attributes in (conforming) XML documents. An element definition (`elemType`) in a DTD consists of a tag name and a structural restriction (`contentSpec`) on the element's children (called a content model). The content model of an element is expressed in a DTD as a regular expression over element tag names. (Regular expressions are one reason DTDs are complex; the other reason is a result of the number of subtle restrictions possible for attributes.) In addition, each element definition in a DTD may also have a set of corresponding attribute definitions (`attDef`), which specify an attribute name and a restriction on the allowable attribute values.

An XML document specifies at most one DTD, which is an assertion that the document conforms to the structure defined by the DTD. Note that an XML document is not required to specify a DTD. By not providing a DTD, the document is considered un-typed, that is, the document does not have a specified schema (assuming no other schema language is used). Thus, the XML data model permits schema-less data.

An XML document is valid with respect to a DTD if the XML document: (1) only contains elements defined within the DTD, (2) each element only contains attributes defined in the DTD and the attributes contain valid values, (3) each required attribute defined in the DTD is present for each corresponding element in the document, and (4) the children nodes of an element are valid with respect to the corresponding element's content model defined in the DTD. Note that an element can be defined in a DTD such that it can contain any content, in which case the above rules are not considered for that particular element's children in a document. Thus, it is possible to mix typed and un-typed data in an XML document.

In a DTD, a content model is designated as either simple (`simpleSpec`), mixed (`mixedSpec`), or as a regular expression (`regExp`). There are three kinds of simple content models: an element is restricted to be empty, that is, it does not contain any children nodes; an element is restricted to contain only character data; or an element is restricted to contain any content including elements not specified in the DTD. A mixed content model allows an element to contain other elements mixed with character data, where the subelements are ordered according to their listing in the content-model specification. And finally, a regular-expression content model specifies the allowable organization of sub-elements for an element. Only elements defined in the DTD can occur as direct children of the element being specified by the regular expression (in contrast to a simple or mixed content model). A regular expression defines a grammar over elements and consists of operations applied to elements and sub-expressions. The allowable operations include concatenation expressed as a list (`listRE`), union expressed as a choice structure (`choiceRE`), and multiple forms of repetition including zero or more (`optManyRE`), zero or one (`optRE`), and one or more (`reqManyRE`).

Attributes can be constrained in a DTD to be required, implied, or fixed. A required attribute must have a value for each corresponding element in the XML document. An implied attribute is optional. And a fixed attribute is required and must contain a specific value (given in the DTD) for each occurrence in the XML document. In addition, the allowable values of an attribute can be specified in a DTD. An attribute can either contain a character data value or be restricted to contain a value from an enumeration, a unique identifier, or a value from an attribute designated as an identifier (called an 'IDREF'). We note that not all combinations of constraints and allowable values are possible, for example, an attribute cannot be defined in a DTD as both fixed and an identifier because each occurrence of an identifier (attribute) must be unique within an XML document.

We note that there are a number of features of DTDs that we do not consider below for simplicity, including attribute lists (in particular, IDREFS), name tokens, notations, entities, and default attributes. Many of these extensions are compiled away by an XML parser and do not significantly change the structural definition of DTDs. Although not included, these additional features have a straightforward definition using the ULD.

The definition below (Data-Model 4) defines both the constructs and constraints of the XML data model with DTDs. The constraints below, for example, for regular expression matching, define part of a validating parser for XML. That is, if all of the constraints are satisfied, the XML document is considered a model of the DTD. Constructing a complete validating parser from the definition below would require additional rules. Namely, we would need to infer the data-instance (*d-inst*) links for attributes and elements. However, these links are trivial to infer because they are based on element-tag and attribute names.

Data-Model 4 (*The XML with DTD data-model constructs and constraints*)

```
construct dtd          = set of elemType
construct elemType     = {hasName->uldString, hasModel->contentSpec,
                           hasAtts->attDefList}
construct contentSpec  = simpleSpec | mixedSpec | regexp
construct simpleSpec   = {isPCData->uldBool, isEmpty->uldBool, isAny->uldBool}
construct mixedSpec    = list of elemType
construct regExp       = listRE | choiceRE | optManyRE | optRE | reqManyRE |
                         elemRE
construct listRE       = emptyRE | fullListRE
construct fullListRE   = {hasRE->regExp, hasTail->listRE}
construct choiceRE     = emptyRE | fullChoiceRE
construct fullChoiceRE = {hasRE->regExp, hasTail->choiceRE}
construct emptyRE      = set of uldValue
construct optManyRE    = {hasRE->regExp}
construct optRE        = {hasRE->regExp}
construct reqManyRE    = {hasRE->regExp}
construct elemRE       = {hasType->elemType}
construct attDefList   = set of attDef
construct attDef       = enumAtt | cDataAtt | idAtt | idRefAtt
construct enumAtt      = reqEnumAtt | impEnumAtt
construct reqEnumAtt   = {hasName->uldString, hasEnum->enumSet}
construct impEnumAtt   = {hasName->uldString, hasEnum->enumSet}
construct enumSet      = set of cdata
construct cDataAtt     = reqCDataAtt | impCDataAtt | fixCDataAtt
construct reqCDataAtt  = {hasName->uldString}
construct impCDataAtt  = {hasName->uldString}
construct fixCDataAtt  = {hasName->uldString, hasFixVal->cdata}
construct idAtt        = reqIdAtt | impIdAtt
construct reqIdAtt     = {hasName->uldString}
construct impIdAtt     = {hasName->uldString}
construct idRefAtt     = reqIdRefAtt | impIdRefAtt | fixIdRefAtt
construct reqIdRefAtt  = {hasName->uldString}
construct impIdRefAtt  = {hasName->uldString}
construct fixIdRefAtt  = {hasName->uldString, hasFixVal->cdata}
construct doc          = {hasRoot->element} conf(domain=*,range=?):dtd
construct element      = {hasTag->uldString, hasAtts->attSet,
                         hasChildren->content} conf(domain=*,range=?):elemtype
construct attSet       = set of attribute
construct attribute    = {hasName->uldString, hasVal->cdata}
                         conf(domain=*,range=?):attDef
construct content      = emptyContent | fullContent
construct fullContent  = {hasNode->node, hasContent->content}
construct emptyContent = set of uldValue
construct node         = pcdata | element
construct cdata        = atomic
construct pcdata       = atomic
```

constraint xml-1 (dtd isNonEmpty).

constraint xml-2 (mixed isNonEmpty).
constraint xml-3 (enumSet isNonEmpty).

constraint xml-4 (hasName keyOf dtd).
constraint xml-5 (hasName keyOf elemType).
constraint xml-6 (hasName keyOf mixed).
constraint xml-7 (hasRE, hasListRE keyOf fullListRE).
constraint xml-8 (hasRE, hasChoiceRE keyOf fullChoiceRE).
constraint xml-9 (hasRE keyOf optManyRE).
constraint xml-10 (hasRE keyOf optRE).
constraint xml-11 (hasRE keyOf reqManyRE).
constraint xml-12 (hasType keyOf elemRE).
constraint xml-13 (hasName keyOf attDefList).
constraint xml-14 (hasName keyOf attSet).
constraint xml-15 (hasNode, hasContent keyOf fullContent).

constraint xml-16 (attDefList always(=) elemType.hasAtts).
constraint xml-17 (enumSet always(=) enumAtt.hasEnum).
constraint xml-18 (attSet always(=) element.hasAtts).
constraint xml-19 (emptyRE always(=) regExp.hasTail).
constraint xml-20 (emptyContent always(=) fullContent.hasContent).
constraint xml-21 (elemType always(∈) dtd).
constraint xml-22 (attDef always(∈) attDefList).
constraint xml-23 (attribute always(∈) attSet).
constraint xml-24 (pcdata always(∈) content).

constraint xml-25 (*If an element is not a root, it is a child of another element*)
c-inst(C, fullContent), C.hasNode:E $\Leftarrow$ c-inst(D, doc), c-inst(E, element)
$\neg$D.hasRoot:E.

constraint xml-26 (*If an element is not a child of another element, it is a root*)
c-inst(D, doc), D.hasRoot:E $\Leftarrow$ c-inst(E, element), $\neg$childElement(E).

childElement(E) $\leftarrow$ c-inst(E, element), c-inst(C, fullContent), C.hasNode:E.

constraint xml-27 (*Only one empty regular expression exists*)
$RE_1=RE_2$ $\Leftarrow$ c-inst(RE_1, emptyRE), c-inst(RE_2, emptyRE).

constraint xml-28 (*The empty regular expression only occurs in the tail of a list or choice*)

$\neg RE_2.hasRE{:}RE_1 \Leftarrow$ c-inst(RE_1, emptyRE), c-inst(RE_2, regExp).

constraint xml-29 (*The empty regular expression has no members*)

$\neg V{\in}RE \Leftarrow$ c-inst(RE, emptyRE), c-inst(V, uldValue).

constraint xml-30 (*An empty content element has no members*)

$\neg V{\in}C \Leftarrow$ c-inst(C, emptyContent), c-inst(V, uldValue).

constraint xml-31 (*Regular expression lists have at least two items*)

$RE_1.hasTail{:}RE_2$, c-inst(RE_2, fullListRE) $\Leftarrow$ c-inst(RE_1, fullListRE),
    $\neg$nestedListRE(RE_1).

nestedListRE(RE_1) $\leftarrow$ c-inst(RE_1, fullListRE), c-inst(RE_2, fullListRE),
    $RE_2.hasTail{:}RE_1$.

constraint xml-32 (*Regular expression choices have at least two items*)

$RE_1.hasTail{:}RE_2$, c-inst(RE_2, fullChoiceRE) $\Leftarrow$ c-inst(RE_1, fullChoiceRE),
    $\neg$nestedChoiceRE(RE_1).

nestedChoiceRE(RE_1) $\leftarrow$ c-inst(RE_1, fullChoiceRE), c-inst(RE_2, fullChoiceRE),
    $RE_2.hasTail{:}RE_1$.

constraint xml-33 (*Choice regular expressions contain unique items*)

$RE{\neq}RE'$ $\Leftarrow$ c-inst(RE_1, fullChoiceRE), $RE_1.hasRE{:}RE$, $RE_1.hasTail{:}RE_2$,
    elementInChoiceRE(RE', RE_2).

elementInChoice(RE', RE_2) $\leftarrow$ $RE_2.hasRE{:}RE'$.
elementInChoice(RE', RE_3) $\leftarrow$ $RE_3.hasTail{:}RE_2$, elementInChoice(RE', RE_2).

constraint xml-34 (*Enumeration sets must have at least two items*)

$CD_1{\in}ES$, $CD_2{\in}ES$, $CD_1{\neq}CD_2$ $\Leftarrow$ c-inst(ES, enumSet).

constraint xml-35 (*Simple content is either pcdata, empty, or any*)

S.isAny:true $\Leftarrow$ c-inst(S, simpleSpec), $\neg$S.isPCData:true, $\neg$S.isEmpty:true.

constraint xml-36 (*Simple content is either pcdata, empty, or any*)

S.isEmpty:true $\Leftarrow$ c-inst(S, simpleSpec), $\neg$S.isPCData:true, $\neg$S.isAny:true.

constraint xml-37 (*Simple content is either pcdata, empty, or any*)

S.isPCData:true $\Leftarrow$ c-inst(S, simpleSpec), $\neg$S.isEmpty:true, $\neg$S.isAny:true.

constraint xml-38 (*One DTD per configuration*)

$D_1=D_2 \Leftarrow$ c-inst(D_1, dtd), c-inst(D_2, dtd).

constraint xml-39 (*One document per configuration*)

$D_1=D_2 \Leftarrow$ c-inst(D_1, doc), c-inst(D_2, doc).

constraint xml-40 (*An element occurs only once in a document*)

$E_1=E_2 \Leftarrow$ c-inst(E, element), E_1.hasChildren:C_1, E_2.hasChildren:C_2, $E \in C_1$, $E \in C_2$.

constraint xml-41 (*An element occurs only once in a document*)

$\neg E \in C \Leftarrow$ c-inst(E, element), c-inst(D, doc), D.hasRoot:E, c-inst(E′, element), E′.hasChildren:C.

constraint xml-42 (*An element occurs only once in a document*)

$\neg$D.hasRoot:E $\Leftarrow$ c-inst(E, element), c-inst(E′, element), E′.hasChildren:C, $E \in C$, c-inst(D, doc).

constraint xml-43 (*An attribute occurs only once in a document*)

$E_1=E_2 \Leftarrow$ c-inst(A, attribute), c-inst(E_1, element), c-inst(E_2, element), E_1.hasAtts:AS_1, E_2.hasAtts:AS_2, $A \in AS_1$, $A \in AS_2$.

constraint xml-44 (*Every configuration with an element has a document*)

c-inst(D, doc) $\Leftarrow$ c-inst(E, element).

constraint xml-conf1 (*If an element has an element type, doc is an instance of the dtd*)

c-inst(T_D, dtd), d-inst(D, T_D) $\Leftarrow$ c-inst(E, element), d-inst(E, T_E), c-inst(D, doc).

constraint xml-conf2 (*If an attribute has an element type, doc is an instance of the dtd*)

c-inst(T_D, dtd), d-inst(D, T_D) $\Leftarrow$ c-inst(A, attribute), d-inst(A, T_A), c-inst(D, doc).

constraint xml-conf3 (*Attribute instances*)

$N_A=N_T \Leftarrow$ c-inst(A, attribute), d-inst(A, T), A.hasName=N_A, T.hasName=N_T.

constraint xml-conf4 (*Enumerated attribute instances*)

$V_A \in ES \Leftarrow$ c-inst(A, attribute), d-inst(A, T), c-inst(T, enumAtt), A.hasVal=V_A, T.hasEnum=ES.

constraint xml-conf5 (*CData fixed attribute instances*)

V_A=V_T ⇐ c-inst(A, attribute), d-inst(A, T), c-inst(T, fixCDataAtt), A.hasVal=V_A,
T.hasFixVal=V_T.

constraint xml-conf6 (*ID attribute instances*)

A_1=A_2 ⇐ c-inst(A_1, attribute), c-inst(A_2, attribute) d-inst(A_1, T), d-inst(A_2, T),
c-inst(T, idAtt), A_1.hasVal=V, A_2.hasVal=V.

constraint xml-conf7 (*IDRef attribute instances*)

c-inst(A_2, attribute), d-inst(A_2, idAtt), A_2.hasVal:V ⇐ c-inst(A_1, attribute), d-inst(A_1, T),
c-inst(T, idRefAtt), A_1.hasVal=V.

constraint xml-conf8 (*IDRef fixed attribute definitions*)

V_A=V_T ⇐ c-inst(A, attribute), d-inst(A, T), c-inst(T, fixIdRefAtt), A.hasVal=V_A,
T.hasFixVal=V_T.

constraint xml-conf9 (*Element instances*)

N_E=N_T ⇐ c-inst(E, element), d-inst(E, T), E.hasTag:N_E, T.hasName:N_T.

constraint xml-conf10 (*Elements must have all required element type attributes*)

E.hasAtts:AS, A_E∈AS, d-inst(A_E, A_T) ⇐ c-inst(E, element), d-inst(E, T), T.hasAtts:AL,
A_T∈AL, requiredAttDef(A_T).

requiredAttDef(A_T) ← c-inst(A_T, reqEnumAtt).
requiredAttDef(A_T) ← c-inst(A_T, reqCDataAtt).
requiredAttDef(A_T) ← c-inst(A_T, reqIdAtt).
requiredAttDef(A_T) ← c-inst(A_T, reqIdRefAtt).

constraint xml-conf11 (*Elements may only have element type attributes*)

A_T∈AL, A_T.hasName:N ⇐ c-inst(E, element), d-inst(E, T), T.hasAtts:AL, E.hasAtts:AS,
A_E∈AS, A_E.hasName:N.

constraint xml-conf12 (*Simple specification*)

|C|=0 ⇐ c-inst(E, element), d-inst(E, T), E.hasChildren:C, T.hasModel:M,
c-inst(M, simpleSpec), M.isEmpty:true

constraint xml-conf13 (*PCData specification*)

V_1=V_2, c-inst(V_1, pcdata) ⇐ c-inst(E, element), d-inst(E, T), E.hasChildren:C,
T.hasModel:M, c-inst(M, simpleSpec), M.isPCData:true, V_1∈C, V_2∈C.

constraint xml-conf14 (*Mixed specification*)

d-inst(V, T), T∈M ⇐ c-inst(E, element), d-inst(E, T), E.hasChildren:C, T.hasModel:M, c-inst(M, mixedSpec), V∈C, ¬c-inst(V, pcdata).

constraint xml-conf15 (*Elements must have the same tagname as the element type*)

E.hasTag:N, T.hasName:N ⇐ c-inst(E, element), d-inst(E, T).

constraint xml-conf16 (*Regular expression specifications*)

¬d-inst(E, T) ⇐ c-inst(E, element), c-inst(T, elemType), E.hasChildren:C, T.hasModel:M, ¬regExpFor(M, C).

regExpFor(M, C) ← c-inst(M, regExp), c-inst(C, content), regExpFor(M, C, C′), c-inst(C′, emptyContent).

regExpFor(M, C, C) ← c-inst(M, emptyRE), c-inst(C, emptyContent).

regExpFor(M, C, C′) ← c-inst(M, elemRE), c-inst(C, fullContent), M.hasType:T, C.hasNode:E, d-inst(E, T), C.hasContent:C′.

regExpFor(M, C, C′) ← c-inst(M, fullChoiceRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′).
regExpFor(M, C, C′) ← c-inst(M, fullChoiceRE), c-inst(C, content), M.hasTail:M′, regExpFor(M′, C).

regExpFor(M, C, C″) ← c-inst(M, fullListRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′), M.hasTail:M″, regExpFor(M, C′, C″).

regExpFor(M, C, C) ← c-inst(M, optManyRE), c-inst(C, content).
regExpFor(M, C, C′) ← c-inst(M, optManyRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′).
regExpFor(M, C, C″) ← c-inst(M, optManyRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′), regExpFor(M, C′, C″).

regExpFor(M, C, C) ← c-inst(M, optRE), c-inst(C, content).
regExpFor(M, C, C′) ← c-inst(M, optRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′).

regExpFor(M, C, C′) ← c-inst(M, reqManyRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′).
regExpFor(M, C, C″) ← c-inst(M, reqManyRE), c-inst(C, content), M.hasRE:M′, regExpFor(M′, C, C′), regExpFor(M, C′, C″).

constraint xml-conf17 (*Document instances*) d-inst(E, T_E), $T_E \in T_D \Leftarrow$ c-inst(D, doc), D.hasRoot:E, d-inst(D, T_D).

3.3.2 RDF Data Models

The Resource Description Framework (RDF) is a graph-based data model for exchanging information over the Web. Information in RDF can be represented in a number of formats, the most widely used is XML. The RDF data model serves as the foundation of the Semantic-Web initiative [11], which aims at creating an infrastructure to enable better search, integration, and discovery of Web-based content by adding semantic, machine processable information to existing Web resources. In the Semantic Web, RDF is seen as a language for exchanging semantic information among agents and other software systems.

The data model for RDF is based on semantic networks [53, 82] in which information is represented using directed graphs with labeled, binary edges and labeled nodes. In RDF, nodes are called resources and edges are called properties. A resource is denoted with a Uniform Resource Indicator (URI), which is a general form of a URL (therefore, a node in an RDF graph can denote a Web-page). Properties in RDF are also represented as URIs. A graph in RDF is represented by a set of triples, which take the form (*property*, *subject*, *object*), where the *subject* and *object* are resources, and the *property* connects the subject (as the domain of the property) to the object (as the range property).

Every assertion in an RDF document is made using triples. To create complex structures such as collections, RDF defines special properties and resources. For example, to represent list data structures, RDF uses the resource `rdf:Seq` (where `rdf` is a namespace and `Seq` is a fragment identifier) and properties `rdf:type` and `rdf:_1`, `rdf:_2`, etc. A resource r is defined as a list, called a *sequence* in RDF, by asserting the triple (`rdf:type`, r, `rdf:Seq`), which is read as "the type of r is a sequence" or equivalently as "r is an instance of a sequence." Elements are added to a sequence using an assertion (`rdf:_1`, r, r'), which is read as "the first item of r is r'." In addition, RDF has a set of special properties for making assertions about triples (called reification), allowing complex hypergraph structures to be defined. Semantic network representations, such as RDF, provide highly extensible systems for modeling information at the cost of clarity and formalism

[82]. In particular, it is easy to build new complex structures from triples by defining special resources and properties. However, doing so often leads to confusion by those who must use the structures because certain properties and resources have a special status or are introduced as a convenience for modeling without being formally defined.

Schema languages for RDF are built using the flexibility offered by RDF. In particular, RDF Schema (RDFS), which is the base schema language for RDF, defines special resources and properties for classifying data. RDFS serves as a very simple description logic [7] that permits the definition of classes and their roles, with universal role restriction. In RDFS, the resource `rdfs:Class` denotes the concept of a class and the resource `rdfs:Property` denotes the concept of a property definition. A property definition includes a domain (through the special property `rdfs:domain`) and a range (through the special property `rdfs:range`) restriction. A particular class c is defined by asserting the triple (`rdf:type`, c, `rdfs:Class`). Objects are created in a similar way by asserting a triple (`rdf:type`, o, c), which is read as "the object o is an instance of class c." A particular property definition (that is, an instance of `rdfs:Property`) can be used as a property in a triple. Finally, RDFS permits subclass (`rdfs:subClassOf`) and subproperty (`rdfs:subPropertyOf`) definitions. Recall that a property specifies a domain and a range class, where a triple that uses the property connects an instance of the domain class to an instance of the range class. A property p_1 can be a subproperty of another property p_2 if the domain class of p_1 is a subclass of the domain class of p_2, and the range class of p_1 is a subclass of the range class of p_2. Thus, a subproperty can restrict a property by requiring a more restrictive domain and range class.

We provide a description of the RDF with RDFS data model below (Data-Model 5). Instead of describing RDF as a standard semantic network representation, we use the ULD to promote RDFS class and property definitions to special constructs in the data model. This approach is closer to the approach of description logics, where schema and data reside at separate levels. Thus, in our representation below, classes and properties are explicitly defined, and can be independently accessed from the data they classify. Note, however, that our representation does not limit the expressibility of RDF with RDFS, because even though a class and a property are separate constructs, they are still considered resources, and thus can still participate in additional assertions. The conformance definitions below were taken from both the RDF Model Theory [41] and the RDF Vocabulary Language [23]. We assume that an RDF processor computes the transitive closure of the subclass and subproperty relationships prior to the creation of the corresponding ULD configuration. Note however that the constraints themselves could be used to infer these additional relationships.

Finally, it is possible to define multiple levels of schema in RDF using the `rdf:type` property. In particular, the `rdf:type` property can be applied without bound and without violating conformance constraints. One use of creating multiple levels of schema is for defining metamodels (also called metaclasses or metaobjects). A metamodel defines an abstract schema for creating conceptual models. A number of systems support metamodeling, such as Telos [59, 43], UML [15, 42], CLOS [44], MOF [61], and Protege [60]. In RDFS, a metamodel is defined as a set of classes and properties with a specific intended semantics. Taking an example from Telos [59], we give a metamodel below using RDF(S) triples for defining conceptual models about document systems.

```
(rdf:type,    DocumentClass, rdfs:Class)
(rdf:type,    AgentClass,    rdfs:Class)
(rdf:type,    SimpleClass,   rdfs:Class)
(rdf:type,    source,        rdfs:Property)
(rdfs:domain, source,        DocumentClass)
(rdfs:range,  source,        AgentClass)
(rdfs:type,   content,       rdfs:Property)
(rdfs:domain, content,       DocumentClass)
(rdfs:range,  content,       SimpleClass)
```

In this example, DocumentClass represents a metaclass for classes concerning documents, for example, instances of DocumentClass could include the classes JournalPaper and ConferencePaper. An AgentClass is a metaclass grouping those classes that serve as the source of a DocumentClass. For example, a Person class might serve as the author of a JournalPaper, where author is a particular source property. The following triples define the JournalPaper and Person classes, along with a particular paper, using RDF(S) (the example is again taken from Telos).

```
(rdf:type,    JournalPaper, DocumentClass)
(rdf:type,    Person,       AgentClass)
(rdf:type,    rdfs:Literal, SimpleClass)
(rdf:type,    author,       source)
(rdfs:domain, author,       JournalPaper)
(rdfs:range,  author,       Person)
(rdf:type,    title,        content)
(rdfs:domain, title,        JournalPaper)
(rdfs:range,  title,        rdfs:Literal)

(rdf:type,    Martian,      JournalPaper)
(rdf:type,    Stanley,      Person)
(rdf:type,    LaSalle,      Person)
(rdf:type,    Wang,         Person)
(author,      Martian,      Stanley)
(author,      Martian,      LaSalle)
(author,      Martian,      Wang)
(title,       Martian,      'The MARTIAN system')
```

Note that multiple levels of schema are used in this example. For example, Martian is an instance of JournalPaper, which is an instance of DocumentClass (which is an instance of `rdfs:Class`).

Data-Model 5 (*The RDF(S) data-model constructs and constraints*)

```
construct resource    = rdfType | simpleRes | complexRes
construct rdfType     = class | property
construct class       = {hasURI->uldURI, hasLabel->uldString}
                        conf(domain=*,range=*):class
construct property    = {hasURI->uldURI, hasDomain->class, hasRange->class}
                        conf(domain=*,range=*):rdfType
construct subClass    = {hasSub->class, hasSuper->class}
construct subProp     = {hasSub->prop, hasSuper->prop}
construct simpleRes   = {hasURI->uldURI} conf(domain=*,range=*):class
construct complexRes  = {hasURI->uldURI, hasVal->complexVal}
construct complexVal  = triple | rdfBag | rdfSeq | rdfAlt
construct triple      = {hasPred->resource, hasSubj->resource, hasObj->objectVal}
construct objectVal   = resource | literal
construct rdfBag      = bag of object
construct rdfSeq      = list of object
construct rdfAlt      = set of object
construct literal     = atomic conf(domain:*,range:*):class
```

constraint rdf-1: (hasURI keyOf resource)
constraint rdf-2: (hasLabel keyOf class)
constraint rdf-3: (hasSuper, hasSub keyOf subClass)
constraint rdf-4: (hasSuper, hasSub keyOf subProp)
constraint rdf-5: (hasPred, hasSubj, hasObj keyOf triple)
constraint rdf-6: (rdfBag isUnique)
constraint rdf-7: (rdfSeq isUnique)
constraint rdf-8: (rdfAlt isUnique)
constraint rdf-9: (rdfAlt isNonEmpty)
constraint rdf-10: (literal always(=) triple.hasObj)

constraint rdf-11: (*Subclass relationship is transitive*)
c-inst(S_3, subClass), S_3.hasSub:C_1, S_3.hasSuper:C_3 $\Leftarrow$ c-inst(S_1, subClass),
c-inst(S_2, subClass), S_1.hasSub:C_1, S_1.hasSuper:C_2, S_2.hasSub:C_2, S_2.hasSuper:C_3.

constraint rdf-12: (*Subproperty relationship is transitive*)
c-inst(S_3, subProp), S_3.hasSub:P_1, S_3.hasSuper:P_3 $\Leftarrow$ c-inst(S_1, subProp),
c-inst(S_2, subProp), S_1.hasSub:P_1, S_1.hasSuper:P_2, S_2.hasSub:P_2, S_2.hasSuper:P_3.

constraint rdf-13: (*Property domains must be subclasses of super property domains*)
c-inst(S_c, subClass), S_c.hasSub:C_1, S_c.hasSuper:C_2 $\Leftarrow$ c-inst(S_p, subProp), S_p.hasSub:P_1,
S_p.hasSuper:P_2, P_1.hasDomain:C_1, P_2.hasDomain:C_2, $C_1 \neq C_2$.

constraint rdf-14: (*Property ranges must be subclasses of super property ranges*)
c-inst(S_c, subClass), S_c.hasSub:C_1, S_c.hasSuper:C_2 $\Leftarrow$ c-inst(S_p, subProp), S_p.hasSub:P_1,
S_p.hasSuper:P_2, P_1.hasRange:C_1, P_2.hasRange:C_2, $C_1 \neq C_2$.

constraint rdf-conf1: (*Classes with literal instances only have literal instances*)
c-inst(L_2, literal) $\Leftarrow$ c-inst(C, class), d-inst(L_1, C), d-inst(L_2, C), c-inst(L_1, literal).

constraint rdf-conf2: (*An instance of a class is also an instance of a super class*) d-inst(X, C_2) ⇐ d-inst(X, C_1), c-inst(C_1, class), c-inst(S, subClass), S.hasSub:C_1, S.hasSuper:C_2.

constraint rdf-conf3: (*An instance of a property is also an instance of a super property*)
d-inst(X, P_2) ⇐ d-inst(X, P_1), c-inst(P_1, property), c-inst(S, subProp), S.hasSub:P_1, S.hasSuper:P_2.

constraint rdf-conf4: (*Triples with constrained properties have conforming subjects and objects*)
d-inst(S, C_d), d-inst(O, C_r) ⇐ c-inst(T, triple), T.hasPred:P, c-inst(P, property), P.hasDomain:C_d, T.hasSubj:S, P.hasRange:C_r, T.hasObj:O.

constraint rdf-conf5: (*Triples are entailed by super properties*)
c-inst(T_2, triple), T_2.hasPred:P_2, T_2.hasSubj:S, T_2.hasObj:O ⇐ c-inst(T_1, triple), T_1.hasPred=P_1, c-inst(S, subProp), S.hasSub:P_1, S.hasSuper:P_2, T_1.hasSubj:S, T_1.hasObj:O.

constraint rdf-conf6: (*Property instances have conforming domain and range values*)
d-inst(P_{d1}, P_{d2}), d-inst(P_{r1}, P_{r2}) ⇐ d-inst(P_1, P_2), c-inst(P_2, property), P_1.hasDomain:P_{d1}, P_1.hasRange:P_{r1}. P_2.hasDomain:P_{d2}, P_2.hasRange:P_{r2}.

3.4 Additional Data Models

This section describes two additional data models for representing structured information. In Section 3.4.1, we describe the Topic Map data model, which like RDF, permits multiple levels of schema and is used to represent information on the Web. In Section 3.4.3, we describe a structured hypertext data model called VIKI, which explicitly permits (and encourages) un-typed and partially typed data. The term *emergent schema* is used with VIKI to describe its operational approach of allowing users to start with un-typed data, and as natural types "emerge," types can be assigned to existing information. Both Topic Maps and VIKI were developed to make it easy for users to navigate unstructured information. A Topic Map provides structures and typing capabilities for navigating information stored externally, for example, in unstructured Web-pages or other documents. VIKI provides structures for organizing arbitrary text-based content that would reside externally, for example, in a Topic Map.

3.4.1 Topic-Map Data Models

Compared to RDF, the goal of the Topic-Map data model [13, 66] is to enable people (as opposed to software agents) to more easily locate and browse digital information. The Topic-Map data model contains structures similar to those found in traditional book indexes, that is, a possibly hierarchical list of subjects, each with one or more references to page numbers, see-also notes, and so on. Due to this similarity, Topic Maps are often referred to as "back-of-the-book" indexes [72] for digital resources. A Topic Map consists of a set of topics, occurrences, associations that connect topics, and scopes.

A topic is a concrete entity that serves to reify (or represent) exactly one abstract subject, for example, a subject may represent a concept in an ontology [13]. Topics contain zero or more base names and occurrences, and can be instances of other topics. A base name provides a convenient textual representation for a topic. An occurrence relates a resource with a topic, where a resource is either external (such as a Web-page), and denoted with a URI, or internal, and represented as a character string. An occurrence is an assertion stating that the topic is relevant to the resource. Finally, the topic instance-of relationship allows topics to be arranged into hierarchies. Both occurrences and associations (see below) can also be typed using topics, however, unlike topics they can have at most one topic type. Figure 3.1 shows a simple Topic Map consisting of a binary association between two topics *m1* and *a1*, where *m1* has two occurrences and *a2* has one occurrence. The remaining topics in Figure 3.1 are used as types.

A topic is said to play a role in an association. Each role in an association can be distinguished by an optional role name (similar to the E-R data model). A role name is assigned using a topic, that is, roles are filled by topics and are distinguished (named) by topics.

A scope is defined in a Topic Map as a special topic for labeling topic names, occurrences, and association roles. A scope establishes a context for applying the labeled item. For example, an *English* scope could be used to label the base names in the Topic Map that are in English. The scope could then be used whenever a user wishes to browse only

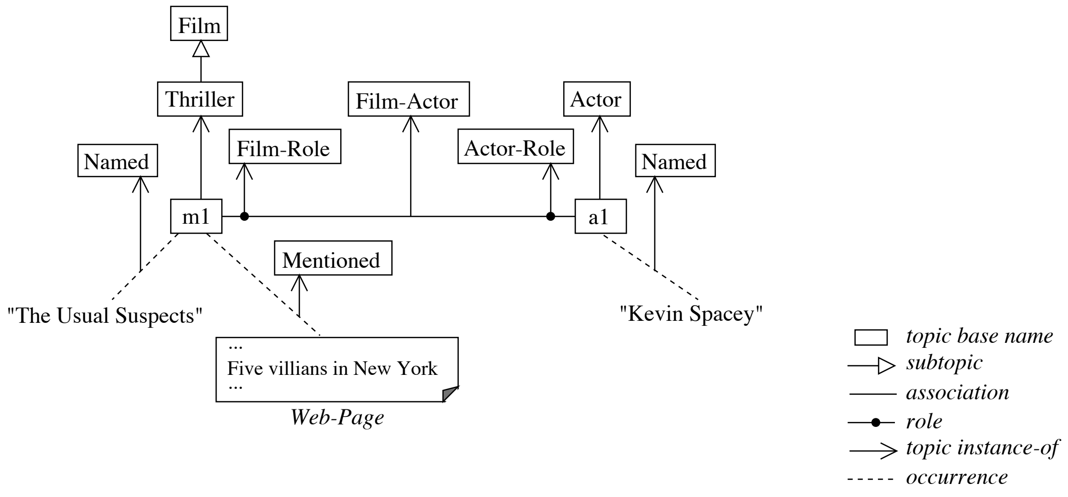

Figure 3.1: An example of a Topic Map represented graphically.

the English portion of the Topic Map. The default scope for a topic map is the *unconstrained scope*. Unless an explicit scope is given, the unconstrained scope is assumed. The *topic naming constraint* asserts that any two topics in the same scope with an identical base name implicitly refer to the same subject, and should be combined into a single topic (by a topic map processor).

The Topic Map data model is an ISO standard [13], and there are a number of versions of the model including an XML interchange format (XTM) [66]. In XTM, subjects and topics are decoupled to simplify the specification of a Topic Map. We we describe a version of the XTM data model below (Data-Model 6). We assume each configuration represents a Topic Map. For convenience, we do not include the additional subject information provided by the full XTM syntax. We note that, while topics and subjects are still considered one-to-one in XTM, by making subjects explicit, it is possible to define a topic in multiple places within a Topic Map and then re-combine the definition into a single topic by an XTM processor. We assume the configuration is constructed after an XTM processor has parsed the Topic Map, and that the parser computes the transitive closure of the subtopic relationship.

Topic-Map data models place few restrictions on class-instance relationships, as reflected in the conformance definitions below. The main restrictions concern transitivity, namely, the class-instance relationship is not transitive, whereas the subtype-supertype relationship is. We include these assumptions below. There are some recent proposals for Topic-Map constraint languages including AsTMa [9] and Topic-Map templates [65] (which are themselves Topic Maps). In practice, if constraints are required in an application, they are currently expressed using conventions introduced by a top-level set of topics that all other topics are assumed to be instances of.

Data-Model 6 (*The XTM data-model constructs and constraints*)

```
construct topic       = {hasId->uldString, hasBNames->baseNames,
                         hasOccurs->occurBag} conf(domain=*,range=*):topic
construct subTopicOf  = {hasSubTopic->topic, hasSuperTopic->topic}
construct baseNames   = set of uldString
construct occurBag    = bag of occurrence
construct occurrence  = {hasValue->occurRes} conf(domain=*,range=?):topic
construct occurRes    = uldString | uldURI
construct association = list of assocMember conf(domain=*,range=?):topic
construct assocMember = {forAssoc->association, hasTopic->topic}
construct memberRole  = {forMember->assocMember, hasRole->topic}
construct bNameScope  = {forTopic->topic, forBName->uldString, hasScope->topic}
construct occurScope  = {forTopic->topic, forOccur->occurrence, hasScope->topic}
construct assocScope  = {forAssoc->association, forMember->assocMember,
                         hasScope->topic}
```

constraint xtm-1: (hasId keyOf topic)
constraint xtm-2: (hasSubTopic, hasSuperTopic keyOf subTopicOf)
constraint xtm-3: (hasValue keyOf occurrence)
constraint xtm-4: (forMember, hasRole keyOf memberRole)
constraint xtm-5: (forTopic, forBName, hasScope keyOf bNameScope)
constraint xtm-6: (forTopic, forOccur, hasScope keyOf occurScope)
constraint xtm-7: (forAssoc, forMember, hasScope keyOf assocScope)

constraint xtm-8: (baseNames always($=$) topic.hasBNames)
constraint xtm-9: (occurBag always($=$) topic.hasOccurs)
constraint xtm-10: (occurence always($\in$) occurBag)
constraint xtm-11: (assocMember always$\bowtie$($\in$) assocMember.forAssoc)

constraint xtm-12: (bNameScope.forBName always$\bowtie(\in)$ bNameScope.forTopic.hasBNames)
constraint xtm-13: (occurScope.forOccur always$\bowtie(\in)$ occurScope.forTopic.hasOccurs)
constraint xtm-14: (assocScope.forMember always$\bowtie(\in)$ assocScope.forAssoc)

constraint xtm-15: (baseNames isUnique)
constraint xtm-16: (association isNonEmpty)

constraint xtm-17: (*Topic naming constraint: topics have disjoint base names in a scope*)
T_1=T_2 $\Leftarrow$ c-inst(S_1,bNameScope), c-inst(S_2,bNameScope), S_1.hasScope:T, S_2.hasScope:T, S_1.forTopic:T_1, S_2.forTopic:T_2, S_1.forBName:B, S_2.forBName:B.

constraint xtm-18: (*An association member is only in its corresponding association*)
$\neg$M$\in A_2$ $\Leftarrow$ c-inst(M,assocMember), M.forAssoc:A_1, c-inst(A_2,association), $A_1 \neq A_2$.

constraint xtm-19: (*The subtopic relationship is transitive*)
c-inst(S_3,subTopicOf), S_3.hasSubTopic:T_1, S_3.hasSuperTopic:T_3 $\Leftarrow$ c-inst(S_1,subTopicOf), c-inst(S_2,subTopicOf), S_1.hasSubTopic:T_1, S_1.hasSuperTopic:T_2, S_2.hasSubTopic:T_2, S_2.hasSuperTopic:T_3.

constraint xtm-conf1: (*Instances of a topic are also instances of the super topics*)
d-inst(T,T_2) $\Leftarrow$ d-inst(T,T_1), c-inst(S,subTopicOf), S.hasSubTopic:T_1, S.hasSuperTopic:T_2.

3.4.2 The VIKI Data Model

The VIKI system [55, 76] is designed as a tool for creating and navigating hypertext documents. Broadly speaking, a hypertext document consists of text-based information organized for navigation. Most hypertext data models consist of nodes and links, where a node contains text and a link connects two or more nodes. For example, the World-Wide Web is considered a distributed hypertext. The data model in VIKI uses object, ordered collection, and composite primitive structures for representing hypertext documents. An object consists of zero or more slots, which are attribute-value pairs. All textual information is stored in object slots. Lists can be nested and can contain objects. And a composite, which is similar to a tuple, groups two or more objects or lists. In VIKI, a user constructs a hypertext by creating basic structures (objects, collections, and composites), by arranging the structures on the screen, and by defining visual characteristics for each structure. For example, a user can create an object in VIKI, choose a shape and color for the object, and place it in a specific position on the screen.

Each basic structure in VIKI may have multiple associated types. For example, an object type can consist of a set of slot names, where a slot name is similar to an attribute name. Objects can contain any number of distinct slots, which consist of a name and a value for the slot. If for a particular object type, an object has a slot for every slot name defined in the type, the object can be assigned the object type. A collection type simply serves as a label, thus, a collection can be assigned any available collection type. A composite type consists of two or more component types (either object or collection types). A composite type, however, is strict. Composite types can only be used to type composite items with identical structure as that prescribed by the type. Namely, a composite structure conforms to a type if it has exactly one subelement for each component type. A type can define visual properties including a shape and a default color for instances. The bottom of Figure 3.2 shows a simple hypertext represented in the VIKI data model. The hypertext includes a composite containing an object and two collections, where each collection contains sub-objects. The top of Figure 3.2 gives a type description for the hypertext.

In VIKI, data structures are decoupled from their physical representation. For example, the same object can be represented multiple times on the screen and each representation can have a distinct set of physical properties. In addition, VIKI does not require the visual properties of basic structures to match the visual properties of their types.

VIKI allows users to explicitly type a particular structure defined in the hypertext. In addition, VIKI offers typing and structuring suggestions. For example, VIKI notifies the user when an item matches a type definition. VIKI also tries to recognize implied structure by suggesting collection and composite structures based on repeating patterns of structures, visual properties, and the proximity of items (via a spatial parser). For example, if objects are placed relatively close together and have similar characteristics (similar physical properties or attributes), then VIKI will suggest a collection for grouping the items.

We describe the VIKI data model constructs and constraints below. We do not include

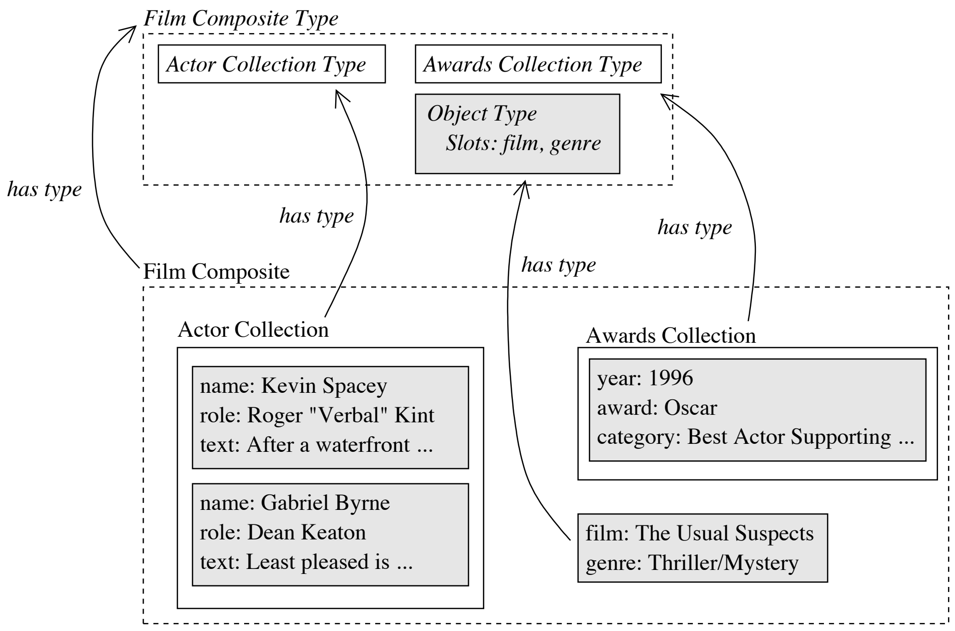

Figure 3.2: An example of a hypertext and its type definition in VIKI.

the visual properties of structures and types (such as font, color, border width, etc.), but instead focus on the data-specific items. (We note that adding visual properties does not fundamentally change the conformance definitions or constructs defined below.) VIKI is an interesting data model to study because it is a mix between object-oriented data models (with objects and collections) and semi-structured data models (such as XML), that allows untyped, partially typed, and fully typed data.

Data-Model 7 (*The VIKI data-model constructs and constraints*)

```
construct objType     = {hasName->uldString, hasSlotDefs->slotDefSet}
construct slotDefSet  = set of slotDef
construct slotDef     = {hasName->uldString}
construct collType    = {hasName->uldString}
construct compType    = {hasName->uldString, hasCompDef->compContDef}
construct compContDef = list of baseType
construct baseType    = objType | collType
construct object      = set of objContent conf(domain=*,range=*):objType
construct objContent  = uldString | slot
construct slot        = {hasName->uldString, hasText->uldString}
construct collection  = {hasName->uldString, hasContent->collContent}
                          conf(domain=*,range=*):collType
construct collContent = list of baseItem
construct baseItem    = object | collection
construct composite   = list of baseItem conf(domain=*,range=*):compType
```

constraint viki-1: (hasName keyOf objType)
constraint viki-2: (hasName keyOf slotDef)
constraint viki-3: (hasName keyOf collType)
constraint viki-4: (hasName keyOf compType)
constraint viki-5: (hasName, hasText keyOf slot)
constraint viki-6: (hasName keyOf object)
constraint viki-7: (hasName, hasContent keyOf collection)

constraint viki-8: (slotDefSet always(=) objType.hasSlotDefs)
constraint viki-9: (slotDef always($\in$) slotDefSet)
constraint viki-10: (compContDef always(=) compType.hasDef)
constraint viki-11: (slot always($\in$) object)
constraint viki-12: (collContent always(=) collection.hasContent)

constraint viki-13: (compContDef isUnique)
constraint viki-14: (object isUnique)
constraint viki-15: (collContent isUnique)
constraint viki-16: (composite isUnique)
constraint viki-17: (compContDef isNonEmpty)
constraint viki-18: (composite isNonEmpty)

constraint viki-19: (*Collections are acyclic*)
$C_1 \neq C_2 \Leftarrow$ c-inst(C_1,collection), unnestCollectionContentOf(C_1,C_2).

unnestCollectionContentOf(C_1,C_2) $\leftarrow$ c-inst(C_1,collection), C_1.hasContent:C_2,
c-inst(C_2,collection).
unnestCollectionContentOf(C_1,C_3) $\leftarrow$ c-inst(C_1,collection), C_1.hasContent:C_2,
c-inst(C_2,collection), unnestCollectionContentOf(C_2,C_3).

constraint viki-20: (*Composites have at least two items*)
$B_1 \in C$, $B_2 \in C$, $B_1 \neq B_2 \Leftarrow$ c-inst(C,composite).

constraint viki-21: (*Composite types have at least two component types*)

$T_1 \in D$, $T_2 \in D$, $T_1 \neq T_2 \Leftarrow$ c-inst(T,compType), T.hasDef:D.

constraint viki-conf1: (*Objects have all of their object-type attributes*)
$S_o \in O$, c-inst(S_o,slot), S_O.hasName:N $\Leftarrow$ d-inst(O,T), c-inst(T,objType),
O.hasSlotDefs:D, $S_t \in D$, S_t.hasName:N.

constraint viki-conf2: (*Composites have the same number of items as their composite types*)
$|C|=L \Leftarrow$ d-inst(C,T), c-inst(T,compType), T.hasCompDef:D, $|D|=L$.

constraint viki-conf3: (*Composite members conform to composite-type members*)
C[I]=B, d-inst(B,T′) $\Leftarrow$ d-inst(C,T), c-inst(T,compType), T.hasCompDef:D, D[I]=T′.

3.5 Properties of Conformance

This section introduces a number of properties for describing conformance, which we use to classify the data models described in this chapter. We consider both structural properties (for example, if conformance is required or optional) and process-oriented properties (for example, if conformance is explicitly created). By *process-oriented* we mean the process by which information is modeled or created within a data model. We conclude this chapter by discussing how the ULD compares to other data models in terms of its use of conformance.

3.5.1 Structural Conformance Properties

Table 3.1 compares the structural conformance properties for the relational, object-oriented, E-R, XML, RDF, Topic Map, and VIKI data models. We describe the conformance properties used in Table 3.1 below. Note that we use the phrase "an instance is typed by a schema" below to mean that a construct instance is an instance of (*d-inst*) another construct instance, the latter instance serving as a type in the schema.

Schema Existence For a data model, existence of a schema is *required* if every configuration of the data model requires a schema definition. Alternatively, existence of a schema is *optional* if a schema is not required (for all configurations of the data model).

Table 3.1: Structural conformance properties of data models.

Data Model	*Schema Existence*	*Individual Participation*	*Extent Restriction*	*Individual Restriction*	*Schema Levels*
Relational	Required	Single	Complete	Complete	Single
O-O	Required	Multiple	Complete	Minimal	Single
E-R	Optional	Multiple	Mixed	Minimal	Single
XML	Optional	Single	Mixed	Complete	Single
RDF(S)	Optional	Multiple	Mixed	Minimal	Collapsed
Topic Map	Optional	Multiple/Single	Mixed	Not Applicable	Collapsed
VIKI	Optional	Multiple	Mixed	Minimal/NA/Complete	Single

Individual Participation A data model requires *single* participation of individuals if instances conform to at most one schema type (when a schema is present). Alternatively, a data model allows *multiple* participation of individuals if instances can conform to multiple schema types.

Extent Restriction By extent, we refer to the implied collection of instances of a construct within a configuration. A data model places a *complete* restriction on an extent if, when a schema exists, all instances in the extent are required to be typed by the schema. Alternatively, a data model allows a *mixed* extent if when a schema exists, instances in the extent can exist without being typed by the schema.

Individual Restriction A data model places a *complete* restriction on individual instances if every instance typed by a schema is precisely typed by the schema, that is, instances do not contain more or less information than defined by their type. A data model places a *minimal* restriction on individual instances if instances typed by a schema are allowed to contain additional information. We note that in some cases, data models do not place any restriction on the structure of typed, individual instances, in which case we say the individual restriction is *not applicable*.

Schema Levels A data model allows at most a *single* level of schema if schema types are not permitted to be instances of other types. Alternatively, a data model allows *multiple* levels of schema if schema types are permitted to be instances of other types. In addition, a data model is *collapsed* if it permits multiple levels of schema using a single construct (that is, it has a recursive conformance relationship to itself) and the construct does not conform to any other constructs (for example, a topic in a Topic Map serves as both data and schema). A data model is *bounded* if it contains a schema construct (that is, a construct that some other construct can conform to) that does not conform to another construct. A bounded data model may or may not have multiple levels of schema. Bounded data models with multiple levels of schema, but without acyclic conformance definitions, have a fixed number of meta-schema levels. (Although possible, we do not consider data models without at least one conformance relationship.)

The schema existence property is reflected in the ULD by the range cardinality restriction on conformance relationships. If any conformance relationship in a data model has a required range constraint (that is, `range` is either 1 or +), schema is required for the data model. Alternatively, schema is considered optional for a data model if there are no required range constraints. For example, in the relational model, the table definition `table = bag of tuple conf(domain=1,range=1):relation` has a range cardinality constraint of 1, thus, the existence of schema is shown as required in Table 3.1. We apply the schema existence property to the data model as a whole, however, it is also possible to consider each conformance relationship separately.

Like schema existence, the individual participation property is also reflected in the ULD by the range cardinality constraint. In particular, if the value of the range constraint is * or +, the data model permits multiple individual participation, and otherwise, permits at most single individual participation. Note that data models with more than one conformance definition (that is, more than one construct with a `conf` assertion) may have different individual participation properties for each such relationship.

The extent and individual restrictions in general are reflected in the ULD using constraint assertion rules. (As a special case, if the existence of schema is required and a particular conformance relationship has a required range constraint, we can infer a complete extent restriction for the schema construct.) Both the extent and individual restrictions are local to data model constructs: data models may have a different value of restriction for each conformance relationship. We note that the extent- and individual-restriction properties are independent. In particular, a data model may require a complete extent restriction and allow a minimal individual restriction. For example, in object-oriented data models, every object must have an associated class (that is, for classes, the data model has a complete extent restriction), however associated objects may have additional attributes, not defined by the class.

The schema-levels property is global for a data model and is reflected in a data model's conformance relationships. In particular, whenever a construct C_1 can conform to a construct C_2, and C_2 can conform to a construct C_3, the data model permits multiple levels of schema (where C_1, C_2, and C_3 are not required to be distinct). When no such C_3 exists, the data model permits only a single level of schema. All data models with a single level of schema are bounded, and collapsed data models are never bounded. Both the bounded and collapsed properties are reflected in the ULD description of a data model by directly analyzing constructs and their conformance definitions.

We note that, in some data models, structural conformance properties may be applied even if there is not an explicit conformance relationship established between constructs. For example, in RDF, the occurrence of a property in a triple places conformance-like restrictions on the subject and object of the triple even though there is not an explicit conformance definition between the constructs. (Note these restrictions are described using conformance assertions in the ULD.)

In Table 3.1, we only list multiple property values for a data model if it contains more than one conformance definition (including conformance-like assertions) and they have different conformance properties. In particular, a topic in a Topic Map can be the instance of multiple topics, but associations and occurrences are allowed to be the instance of at most one topic. In VIKI, an object type specifies a minimal individual restriction, a composite type requires a complete individual restriction, and the individual restriction for a collection type is not applicable.

Finally, we note that XML includes the special *any* content model, which enables a mixed extension restriction, that is, not all elements and attributes are required to be typed by a DTD. However, when a DTD is specified, every element and attribute not associated with an *any* content model must be completely constrained by the DTD.

3.5.2 Process-Oriented Conformance Properties

Table 3.2 compares the process-oriented conformance properties for the relational, object-oriented, E-R, XML, RDF, Topic Map, and VIKI data models. As shown, we consider how conformance is established, and in particular, how the *d-inst* relationship is created.

Conformance is established *implicitly* in a data model if it is generated as a side-effect of creating data or schema. For example, in the relational model, the creation of a relation automatically generates the relation's corresponding table. Similarly, in many object-oriented models, an object is created using a class constructor, which establishes conformance from the object to the class (and to the appropriate superclasses). Note that data models that require conformance to be established *implicitly* are *schema-first*, because schema must exist prior to creating associated data.

We say conformance is *inferred* if it is computed. For example, in the original E-R data model [27], the type of an entity is computed based on its inherent structure. Often, inferred conformance definitions are computable upon request, for example, by initiating a parser (as in the case of XML or RDF).

Finally, conformance is established *explicitly* if the data model offers a mechanism for declaring conformance explicitly. For example, to create conformance in RDF(S), a triple must be asserted using the `rdf:type` property. We note that data models that allow conformance to be established explicitly must permit data to be defined prior to schema, that is, the data model is *schema-later*, and generally permit more flexible conformance definitions. For example, RDF, Topic Maps, and VIKI each require conformance to be established explicitly.

3.5.3 The ULD Conformance Properties

Here, we compare the ULD to other data models in terms of conformance. Even though the ULD is designed as a meta-data-model (that is, to represent other data models) and not as a general purpose data model, generally speaking, the ULD is another data model—it is a structured representation for data—that can represent constructs as well as schema and data.

Table 3.2: Process-oriented conformance properties of data models.

Data Model	*How is conformance established?*	
Relational	Implicitly	When relations are defined
O-O	Implicitly	When an object is created
E-R	Inferred	An entity's type is computed from its structure
XML	Inferred	An XML processor checks conformance
	Explicitly	A document declares a document type
RDF(S)	Inferred	Additional conformance computed from subclass and subproperty
	Explicitly	When the `rdf:type` property is used
Topic Map	Explicitly	Items are typed by stating their topic type
VIKI	Inferred	Users can choose to accept the systems typing suggestions
	Explicitly	Users can explicitly state the type of an item

For the structural conformance properties, the ULD requires the existence of schema (every data instance is an instance of a construct, and every construct is an instance of a construct type), permits multiple individual participation (via the union construct), requires complete extent restriction, and requires complete individual restriction. More importantly, the ULD is exactly a bounded data model that permits multiple levels of schema.

Note that none of the data models we consider are both bounded and permit multiple levels of schema. By being a bounded data model, the ULD can distinguish and enforce the construct type, construct, and data-instance levels and provide a clear definition for each level. Additionally, by allowing multiple levels of schema (according to the *d-inst* relationship), the ULD can represent data models that permit optional and multiple levels of schema.

Finally, in terms of process-oriented properties, conformance in the ULD is established implicitly (when data is created) using the *ct-inst* and *c-inst* relationship, and explicitly for the *d-inst* relationship. Note that, as defined in Chapter 2, the *d-inst* relationship is (explicitly) defined when an object is created using the *c-inst* relationship.

3.6 Summary

In this chapter, we used the ULD to define a number of data models, each with distinct constructs and conformance properties. We also extended the ULD constraint language with constraint macros, which offer short-hand notations for expressing common data-model constraints. Each constraint macro has a corresponding expression in the ULD constraint language. (Thus, constraint macros can be executed using the ULD constraint language.) We used constraint macros and the ULD constraint language to specify the inherent constraints of each data model presented in this chapter. We also defined a classification scheme for data models based on conformance. Finally, we compared the conformance properties of the ULD to other data models.

The definitions of the data models presented in this chapter were taken from various sources. The relational and E-R data models are standard and are formalized in a number of textbooks. However, there are many slightly different object-oriented data-models, each with distinct features. We chose an early version of the O_2 data model, which offers many of the most common features found in object-oriented data models. The XML, RDF, and Topic-Map data models are being defined as standards. There has been some work aimed at formalizing certain features of XML and RDF. For instance, a partial axiomatization has been developed for RDF Schema [41], which we used directly to define conformance constraints. The VIKI data model was designed for use in a hypertext tool, and is described informally in research papers. In general, we believe the ULD can benefit a data-model engineer by offering a mechanism to formally define the constructs and constraints of a data model. By formalizing a data model, a data-model engineer can provide a clear, concise, and unambiguous specification of a data model.

Our goal in this chapter was to show that the ULD can be used to give complete descriptions of a wide range of data models. As future work, we believe that it would be useful to develop techniques to help data-model engineers verify that data-model constraints are consistent and correct (that is, correct with respect to either a formalization of a data model, or to an engineer's understanding of the data model). For example, it may be possible to generate sample schema and data instances from the ULD description of the data model, and ask the data-model engineer if the samples are valid. If a sample is not valid, we can infer that the constraints are not restrictive enough. Alternatively, it may be useful to generate samples that violate data-model constraints, and ask the data-model engineer if the samples are valid. If such a sample is considered valid, we can infer that the ULD constraints are too restrictive. The challenge of this approach is to select the appropriate set of test cases (that is, samples) from the ULD definition of the data model to determine whether the data-model constraints are correct.

For the XML data model, we used this approach to verify that the constraints given are accurate. In particular, for our tests, we used Prolog to represent and execute the XML data-model constraints. We also (manually) defined a set of test documents for a specific DTD. The test documents were designed to highlight specific cases of the data-model constraints. For example, we defined both valid and invalid documents to test each regular-expression operator. In general, we believe the problem is similar to software testing, which may offer useful techniques to determine whether data-model constraints are correctly specified.

Chapter 4

Transformation

This chapter describes a language for data model, schema, and data transformation. The language extends the ULD query language defined in Chapter 2 and can express flexible mappings that access multiple levels of information including data model, schema, and instance data. We begin in Section 4.1 by describing the transformation language. In Section 4.2, we use the language to describe mappings from the E-R and XML data models to the relational data model, a schema mapping for the example schemas in Chapter 1, and a mapping that describes an RDF serialization in XML. Finally, in Section 4.3, we sketch approaches for reasoning with mappings expressed in the transformation language. In particular, we give examples that use constraints to check for well-formed transformations and to elaborate partial mappings.

4.1 The ULD Transformation Language

The ULD transformation language exploits stratified Datalog$^\neg$ to express mappings between heterogeneous information sources. The ULD transformation language is similar to the Well-founded Object Language (WOL) [35] in that it is a declarative, logic-based language for expressing mappings. The WOL language, however, is designed specifically for a restricted object-oriented data model to express schema mappings, whereas our transformation language leverages the ULD to express a wide range of transformations.

A ULD *mapping rule* is expressed against source and target configurations. For example, the following rule converts E-R entity types to relations in the relational data model.

rel/c-inst(T, relation) $\Longleftarrow$ er/c-inst(T, entityType).

The rule is read "if T is an entity-type identifier in the *er* configuration, then T is a relation in the *rel* configuration," where *er* is the source configuration and *rel* is the target configuration. We use the special symbol $\Longleftarrow$ to denote a mapping rule: the body of the rule is on the right side of the $\Longleftarrow$ symbol and the head of the rule is on the left side of the $\Longleftarrow$ symbol. The mapping above states only that entity-type identifiers in a source configuration are mapped to relation identifiers in a target configuration. Additional rules are needed to transform entity-type names to relation names, entity-type attributes to relation attributes, to convert primary keys, and so on.

A *transformation* consists of one or more mapping rules and an optional set of rules that define intensional predicates (used by the mapping rules). For present purposes, we assume there is exactly one source and one target configuration for a transformation. We also assume that a target configuration contains a description of its data model prior to the execution of a transformation. Target configurations may also contain additional schema and instance data.

A mapping rule takes the following form, where $n \geq 1$, $l \geq 0$ and m is the rule name.

mapping m:
h_1, h_2, ..., $h_n \Longleftarrow b_1$, b_2, ..., b_l.

The head of a mapping rule consists of one or more positive ULD formulas h_1 to h_n expressed over a target configuration. As in Datalog¬, the body of a rule consists of zero or more positive or negative extensional formulas b_1 to b_l expressed against a source configuration. Optionally, b_1 to b_l can also contain formulas expressed against a target configuration as well as intensional formulas defined in the transformation. The syntax for mapping rules is similar to ULD constraints, however, mapping rules have a different interpretation. Namely, a mapping rule is equivalent to a rule expressed using stratified Datalog¬ extended to allow more than one formula in the head of the rule.

When a mapping rule is executed, each formula in the head of the rule is added to (or asserted in) a target configuration. In other words, a mapping rule specifies one or more updates to target configurations. The output of a transformation is the updated target configuration that results from computing the fixpoint of the corresponding mapping and intensional rules.

In the previous example, a new relation is constructed in the target configuration for every entity type in the source configuration. As another example, the following mapping rule adds the name and an (empty) attribute list to every generated relation in *rel* (the target configuration).

> *rel*/T.hasName:N, *rel*/c-inst(N, uldString), *rel*/T.hasAtts:AS,
> *rel*/c-inst(AS, attList) $\Longleftarrow$
>   *er*/c-inst(T, entityType), *er*/T.hasName:N, *er*/T.hasAtts:AS.

As shown, the head of this rule contains multiple formulas. Each formula in the head of the rule is asserted in the target configuration such that every variable is replaced with its corresponding variable assignment. (A new variable assignment is obtained in each evaluation of the body of the rule.) We allow multiple formulas in the head of a mapping rule primarily for convenience. However, there are certain mapping rules (which we describe in more detail below) that cannot be expressed in standard Datalog, that is, when only one formula is allowed in the head of a mapping rule.

The configuration name for a formula can be dropped in a mapping rule when default source and target configurations are specified. In particular, formulas in the body of a mapping rule without explicit configuration modifiers are matched against formulas in the default source configuration, and formulas in the head of the rule are inserted into the default target configuration. Thus, the previous example can be equivalently expressed as follows, where the default source configuration is *er* and the default target is *rel*.[1]

> T.hasName:N, c-inst(N, uldString), T.hasAtts:AS, c-inst(AS, attList) $\Longleftarrow$
>   c-inst(T, entityType), T.hasName:N, T.hasAtts:AS.

So far, we have assumed entity-type identifiers do not conflict with identifiers in the target configuration. That is, each entity-type identifier in the source configuration is

[1]Because we assume a single source and target configuration for the transformations presented in this chapter, we typically drop the configuration modifier in mapping rules.

added directly to the target configuration as an instance of a relation construct. If a source identifier is already used in the target configuration, the resulting configuration may not be well-formed. (For example, the identifier could become an instance of more than one non-union construct in the target.) The following rule guarantees that non-conflicting identifiers are added to the target. The rule is read "if T is an entity type in the source, generate a new identifier R in the target such that R is a relation."

c-inst(R, relation) $\Longleftarrow$ c-inst(T, entityType).

This rule introduces an *implicit* mapping between identifiers T and generated identifiers R. If the rule contained a second formula in the head that also used R, the same variable assignment for R is used in both formulas (thus, the same value of R is used in both formulas to update the target). For example, the following rule uses two head formulas to connect the name of an entity type to the name of the corresponding relation. Note that an equivalent Datalog representation does not exist for this mapping rule. The mapping rule cannot be divided into an equivalent set of Datalog rules that contain exactly one formula in the head because R is assumed to be assigned the same value in both formulas.

c-inst(R, relation), R.hasName:N $\Longleftarrow$ c-inst(T, entityType), T.hasName:N.

We provide a mechanism to define *explicit* mappings between identifiers using the identifier-creation operator $\mathcal{K}$, which generates new identifiers using existing source-configuration identifiers (or using labels, which we describe below). The identifier-creation operator $\mathcal{K}$ is a Skolem function [39]. An advantage of using a Skolem function to create identifiers is it provides a mechanism to retrieve the generated identifiers in subsequent rules. To illustrate, consider the following mapping rule that implicitly generates new identifiers.

R.hasName:N, c-inst(N, uldString), R.hasAtts:AL, c-inst(AL, attList) $\Longleftarrow$
    c-inst(T, entityType), T.hasName:N, T.hasAtts:AS.

Once this rule is executed for a particular entity-type T, we can no longer add attributes of T to the corresponding relation R in subsequent rules. The connection between T and

R (and similarly for AS and AL) is lost. The following rule is an equivalent mapping that uses $\mathcal{K}$ to define the explicit connection between T and R (and AS and AL). We describe the syntax of the $\mathcal{K}$ operator below.

$$\begin{aligned}&\text{R.hasName:N, c-inst(N, uldString), R.hasAtts:AL, c-inst(AL, attList)} \Longleftarrow\\&\qquad \text{c-inst(T, entityType), T.hasName:N, T.hasAtts:AS, } \mathcal{K}^{T}(\text{R}),\ \mathcal{K}^{AS}(\text{AL}).\end{aligned}$$

In this example, the values (that is, variable assignments) of T are used to create identifiers that serve as variable assignments for R. (Note that T must be ground when $\mathcal{K}$ is called.) Given the same value for T, $\mathcal{K}^{T}$(R) always unifies R with the same identifier. Other mapping rules can now obtain the relation (or attribute list) identifier generated from the original entity type (or attribute set, respectively), as shown in the following mapping rule, which converts entity-type attributes to relation attributes.[2]

$$\begin{aligned}&\text{A}_r\text{∈AL, c-inst(A}_r\text{, attribute), A}_r\text{.hasName:N, c-inst(N, uldString),}\\&\text{A}_r\text{.hasDomain:D} \Longleftarrow\\&\qquad \text{c-inst(T, entityType), T.hasAtts:AS, A}_e\text{∈AS, A}_e\text{.hasName:N,}\\&\qquad \text{A}_e\text{.hasDomain:D, } \mathcal{K}^{AS}(\text{AL}),\ \mathcal{K}^{A_e}(\text{A}_r).\end{aligned}$$

In general, the identifier creation operator takes the form $\mathcal{K}_L^{V_1,V_2,\ldots,V_n}$(V) for $n \geq 0$, where V_1 to V_n are (ground) variables representing source-configuration identifiers, L is an optional constant label, and V is a variable that is unified with the generated identifier. When V is bound, the call to $\mathcal{K}$ returns true if V can be unified with the value of the created identifier, and false otherwise. The $\mathcal{K}$ operator can equivalently be written as the Skolem function $\mathcal{K}(V_1, ..., V_n, L)$. Thus, given values for V_1 to V_n and L, $\mathcal{K}$ always returns the same value V.[3]

[2]Note that even with $\mathcal{K}$, there are still cases where multiple formulas in the head a mapping rule are needed. In particular, multiple formulas are needed if the body of the rule depends on information in the target configuration and the rule generates new information in the target. An example of this case is shown by the mapping rule `rdfxml-3` in Section 4.2.4.

[3]We implement the $\mathcal{K}$ operator in Prolog (see Section 6.1) using the binary predicate `idgen`, where the first argument is a list of terms T, and the second argument is the generated identifier. The list of terms T is used to hold the ground, source-configuration identifiers (V_1 to V_n) and the constant label (L). For example, the expression $\mathcal{K}_{attList}^{AS}$(AL) is represented in Prolog as `idgen([AS, ‘attList’], AL)`.

The expression $A_r \in AL$ in the head of the previous example inserts A_r into the list AL. In general, an expression $X \in C$ in the head of a mapping rule serves to insert the value X into the collection C for each unique evaluation of the body of the rule. For lists, we assume the expression $X \in C$ appends X to the end of the list. It is also possible to state the index of X in the list C explicitly using the expression $C[I] = X$, which also adds X to C.

In addition to the $\mathcal{K}$ operator, we also permit other special-purpose functions. For example, mathematical operations such as addition as well as concatenation of strings (denoted using the '$\cdot$' operator) are used in the examples of Section 4.2. We also permit user-defined functions, which are often used to resolve differences in data. For example, a user-defined function can be used to convert full names in last-name-first format to full names in first-name-first format. As in Datalog, we require function parameters to be ground when a function is called, that is, every parameter is either a constant or a variable unified with a constant through a variable binding.

Finally, we note that it is possible to create non-terminating mapping rules using the ULD transformation language. For instance, consider the following rule (which does not terminate), where r is the name of the target configuration and c is a construct.

$$r/\text{c-inst}(\text{Y}, c) \Longleftarrow r/\text{c-inst}(\text{X}, c),\ \mathcal{K}^X(\text{Y}).$$

This rule does not have a fixpoint: Each time the body is evaluated, a new identifier is generated, and the formula inserted into the target causes the body of the rule to have a new evaluation. The rule results in infinite recursion. (Note that we can restrict the domain of $\mathcal{K}$ to source-configuration variables, in which case, the rule above would not be well-formed.)

4.2 Transformation Examples

This section defines a set of common transformations using the ULD transformation language. We start by describing an E-R to relational transformation. We divide the transformation into a set of schema and data mappings. By applying both transformations, the schema and instance data in an E-R configuration is converted to a relational schema and a corresponding database in the target configuration. Alternatively, the schema transformation can be applied in isolation, which is what is typically done in conceptual modeling approaches. We then give two standard XML-to-relational transformations. The first transformation converts XML instance data directly to a relational database, without the use of an XML DTD. The second transformation requires the XML source to have a DTD, which is converted to a relational schema, and the XML instance data is converted to a corresponding database for the schema. Next, we describe a schema transformation from the XML DTD of Figure 1.2 to the RDF Schema of Figure 1.3. The transformation can be used to convert XML documents that conform to the DTD in Figure 1.2 to RDF instances that conform to the RDF Schema in Figure 1.3. Finally, we describe a transformation from RDF to XML that defines an XML interchange format for RDF similar to the standard RDF XML syntax [47]. The advantage of such a transformation is that the mapping between the data model and its serialization syntax (that is, the syntax used to interchange schema and instance data in the data model) is made explicit.

We note that none of the transformations described here can be fully expressed in traditional meta-data-model frameworks. In particular, the transformations use a mix of data-model, schema, and instance data to introduce conventions for representing a source configuration in a target data model.

4.2.1 E-R to Relational Transformations

The E-R to relational schema mapping below is similar to the mapping given by Elmasri and Navathe [38]. Each entity type is converted to a relation (er-rel-1) and the attributes of entity types are mapped to corresponding relation attributes (er-rel-2). Similarly, each relationship type is mapped to a relation (er-rel-4) and relationship-type attributes are mapped to relation attributes (er-rel-5).

We simplify the transformation defined by Elmasri and Navathe as follows. Instead of using the primary key of an entity type as the primary key for the corresponding relation, we introduce a new relation attribute called "id" (er-rel-3). The id-attribute serves as a key for the relation. Each relation that is created from an E-R relationship type contains one attribute for each role of the relationship type (er-rel-6). A foreign key is also created for each role attribute to the corresponding id-attribute. (We note that it is possible to define a mapping using the ULD transformation language to convert entity-type primary keys to relation primary keys, but for exposition, we choose the simpler id-attribute approach below.)

Transformation 1 (*E-R-to relational schema transformation*)

mapping er-rel-1: (*Entity types become relations*)
c-inst(R, relation), R.hasName:N, c-inst(N, uldString) $\Longleftarrow$
c-inst(T, entityType), T.hasName:N, $\mathcal{K}^{T}_{relation}$(R).

mapping er-rel-2: (*Entity-type attributes become relation attributes*)
R.hasAtts:AL, c-inst(AL, attList), c-inst(A_r, attribute), $A_r \in$AL, A_r.hasName:N, c-inst(N, uldString) A_r.hasDomain:D $\Longleftarrow$
c-inst(T, entityType), T.hasAtts:AS, $A_e \in$AS, A_e.hasName:N, A_e.hasDomain:D, $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{AS}_{attList}$(AL), $\mathcal{K}^{A_e}_{attribute}$($A_r$).

mapping er-rel-3: (*Create a key for each relation*)
c-inst(A, attribute), A.hasName:'id', c-inst('id', uldString), A.hasDomain:'uldString', A$\in$AL, c-inst(P, pKey), P.forType:R, P.keyAtts:PL, c-inst(PL, pKeyAttList), A$\in$PL $\Longleftarrow$
c-inst(T, entityType), T.hasAtts:AS, $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{AS}_{attList}$(AL), $\mathcal{K}^{T}_{id}$(A), $\mathcal{K}^{T}_{pkey}$(P), $\mathcal{K}^{T}_{pkeyattlist}$(PL).

mapping er-rel-4: (*Relationship types become relations.*)
c-inst(R, relation), R.hasName:N, c-inst(N, uldString) $\Longleftarrow$
c-inst(T, relType), T.hasName:N, $\mathcal{K}^{T}_{relation}$(R).

mapping er-rel-5: (*Relationship attributes become relation attributes.*)
R.hasAtts:AL, c-inst(AL, attList), c-inst(A_r, attribute), $A_r \in$AL, A_r.hasName:N, c-inst(N, uldString) A.hasDomain:D $\Longleftarrow$
c-inst(T, relType), T.hasAtts:AS, $A_e \in$AS, A_e.hasName:N, A_e.hasDomain:D, $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{AS}_{attList}$(AL), $\mathcal{K}^{A_e}_{attribute}$($A_r$).

mapping er-rel-6: (*Relationship roles become foreign key attributes.*)
c-inst(A, attribute), A$\in$AL, A.hasName:N, c-inst(N, uldString), A.hasDomain:'uldString', c-inst(F, fKey), fKey.forRel:R_r, fKey.toRel:R_e, fKey.keyAtts:FL, c-inst(FL, fKeyAttList), A$\in$FL $\Longleftarrow$

c-inst(T_r, relType), T_r.hasAtts:AS, T_r.hasRoles:RL, RL[I]=R, R.hasType:T_e,
N=‘role’ · I, $\mathcal{K}^{R,I}_{attribute}$(A), $\mathcal{K}^{R,I}_{fkey}$(F), $\mathcal{K}^{R,I}_{fkeyattlist}$(FL), $\mathcal{K}^{AS}_{attList}$(AL), $\mathcal{K}^{T_r}_{relation}$($R_r$),
$\mathcal{K}^{T_e}_{relation}$($R_e$).

The E-R to relational data mapping given below stores each entity and relationship as a tuple in one or more relations. For example, every entity that has an entity type is represented as a tuple in the entity type's corresponding relation. An entity or relationship with more than one type is transformed into multiple tuples, one for each corresponding type.

The data transformation converts only those entities and relationships with a corresponding entity and relationship type. Thus, the mapping selects only the E-R items that are covered by schema. We could additionally represent those entities and relationships that do not have corresponding schema by defining two default relations, one for entities and one for relationships. For example, the default entity relation would consist of three attributes, one to store the entity identifier, one to store a property name, and one to store a property value. Thus, the table would consist of one tuple for every entity property, where each such tuple has the same value for the id-attribute.

Note that, in the mapping rules er-rel-8 and er-rel-13, we assume for simplicity that both configurations have the same atomic types. Otherwise, we would need to add additional formulas to the head of each rule to define the domain type in the source.

Transformation 2 (*E-R-to-relational data transformation*)

mapping er-rel-7: (*Every entity relation has a corresponding table.*)
c-inst(B, table), d-inst(B, T) $\Longleftarrow$
c-inst(T, entityType), $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{T}_{table}$(B).

mapping er-rel-8: (*Entities become tuples.*)
c-inst(P, tuple), P∈B, d-inst(P, R) $\Longleftarrow$
c-inst(E, entity), d-inst(E, T), $\mathcal{K}^{E,T}_{tuple}$(P), $\mathcal{K}^{T}_{table}$(B), $\mathcal{K}^{T}_{relation}$(R).

mapping er-rel-9: (*Entity attributes become relation attributes.*)
P[I]=V, c-inst(V, C) $\Longleftarrow$
c-inst(E, entity), d-inst(E, T), E.hasProps:P_e, P∈P_e, P.hasName:N, P.hasVal:V,
c-inst(V, C), T.hasAtts:AS, $\mathcal{K}^{AS}_{attList}$(AL), *rel*/AL[I]=A, *rel*/A.hasName:N,

$\mathcal{K}^{E,T}_{tuple}$(P).

mapping er-rel-10: (*Create a key value for each entity tuple.*)

P[I]=V, c-inst(V, uldString) $\Longleftarrow$

c-inst(E, entity), d-inst(E, T), T.hasAtts:AS, $\mathcal{K}^{AS}_{attList}$(AL), *rel*/AL[I]=A, *rel*/A.hasName:'id', $\mathcal{K}^{E,T}_{id}$(V), $\mathcal{K}^{E,T}_{tuple}$(P).

mapping er-rel-11: (*Every relationship has a corresponding table.*)

c-inst(B, table), d-inst(B, R) $\Longleftarrow$ c-inst(T, relType), $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{T}_{table}$(B).

mapping er-rel-12: (*Relationships become tuples.*)

c-inst(P, tuple), P$\in$B, d-inst(P, R$'$) $\Longleftarrow$

c-inst(R, rel), d-inst(R, T), $\mathcal{K}^{T}_{table}$(B), $\mathcal{K}^{R,T}_{tuple}$(P), $\mathcal{K}^{R}_{relation}$(R$'$)

mapping er-rel-13: (*Relationship attributes become relation attributes.*)

P[I]=V, c-inst(V, C) $\Longleftarrow$

c-inst(R, rel), d-inst(R, T), R.hasProps:P$_e$, P$\in$P$_e$, P.hasName:N, P.hasVal:V, c-inst(V, C), T.hasAtts:AS, $\mathcal{K}^{AS}_{attList}$(AL), *rel*/AL[I]=A, *rel*/A.hasName:N, $\mathcal{K}^{R,T}_{tuple}$(P).

mapping er-rel-14: (*Relationship roles become relation attributes.*)

P[I]=V, c-inst(V, uldString) $\Longleftarrow$

c-inst(R, rel), d-inst(R, T$_r$), T$_r$.hasAtts:AS, T$_r$.hasRoles:RL, R.associates:EL, EL[I$_e$]=E, AS[I$_e$]=T$_e$, $\mathcal{K}^{AS}_{attList}$(AL), $\mathcal{K}^{R,I_e}_{attribute}$(A), *rel*/AL[I]=A, $\mathcal{K}^{E,T_e}_{id}$(V), $\mathcal{K}^{R,T_r}_{tuple}$(P).

4.2.2 XML to Relational Transformations

We consider two XML to relational transformations. The first is given below (Transformation 3) and is called the "edge" approach [77]. This transformation does not consider schema information from the XML configuration and instead transforms the XML document directly to tuples in a pre-defined relational schema.

In the edge approach, a single relational table (called the *edge* table) is used to store an XML document. The relation for the edge table contains five attributes. To illustrate, consider the XML example from Figure 1.2 represented as a tree in Figure 4.1 and in an edge table in Figure 4.2. As shown, each element, attribute, and PCDATA value is represented as a distinct node. Each node in the XML document, except PCDATA, is

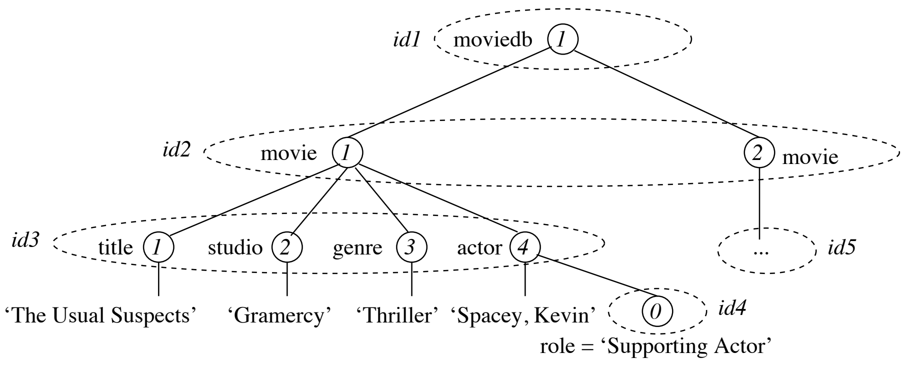

Figure 4.1: The edge-based, tree representation of the XML document of Figure 1.2. Nodes are annotated with their ordinal number and levels are annotated with their *sourceId* value.

mapped to a tuple in the edge table. The first attribute of the edge table, called *SourceId*, stores an identifier that uniquely identifies the parent node. For example, the root node (*moviedb*) of Figure 4.1 has identifier *id1*, the children of *moviedb* have the identifier *id2*, the children of the first child of *moviedb* have the identifier *id3*, the children of the second child of *moviedb* have the identifier *id5*, and so on. The second attribute of the edge table stores the node's label, which is either an element name or an attribute name (called *Tag* [77]). The third attribute stores the order (*Ordinal*) of a node relative to its parent node. For attributes, the ordinal value is always zer and for the root element the ordinal value is one. The fourth attribute (*TargetId*) stores the *SourceId* for the children of the node or 'NULL' if the node does not have any children. Finally, the fifth attribute (*Data*) stores the PCDATA value, the attribute value, or 'NULL' if neither exists. Note that the *Data* attribute is an example of inlining [74, 77].

The mapping rules `edge-1` through `edge-6` below create the pre-defined relational schema for the edge transformation. The last mapping rule (`edge-7`) uses the *findEdge* intensional predicate to generate tuples, which are stored in the edge schema.

Our motivation for including the XML edge-based mapping is to highlight the flexibility of the ULD transformation language. In particular, the transformation expressed in the ULD works regardless of whether a schema is defined for the XML configuration. Existing meta-data-model approaches are based on the assumption that schema is always defined

EdgeTable

SourceId	*Tag*	*Ordinal*	*TargetId*	*Data*
id1	moviedb	1	id2	‘NULL’
id2	movie	1	id3	‘NULL’
id2	movie	2	id5	‘NULL’
id3	title	1	‘NULL’	‘The Usual Suspects’
id3	studio	2	‘NULL’	‘Gramercy’
id3	genre	3	‘NULL’	‘Thriller’
id3	actor	4	id4	‘Spacey, Kevin’
id4	role	0	‘NULL’	‘Supporting Actor’
id5	title	1	‘NULL’	‘Meet the Parents’
id5	studio	2	‘NULL’	‘Universal’
id5	genre	3	‘NULL’	‘Comedy’
id5	actor	4	id6	‘De Niro, Robert’
id6	role	0	‘NULL’	‘Leading Actor’

Figure 4.2: The result of applying the edge transformation.

for a source. Thus, they cannot express such data-only transformations because access to data can occur only through schema structures.

Finally, the edge-based transformation maps XML data directly to relational data using an agreed-upon convention. We note that, in general, conventions are often introduced in transformations among heterogeneous information sources due to the differences in the constructs of the data models. In the edge approach, XML information is stored in the relational data model using a predefined table to represent an ordered tree, because the relational data model does not have constructs for explicitly representing hierarchical information.

Transformation 3 (*An XML-to-relational edge transformation*)

mapping edge-1: (*Construct the edge table.*)

c-inst(edgeTbl, relation), edgeTbl.hasName:‘EdgeTable’, c-inst(‘EdgeTable’, uldString), c-inst(edgeTblAtts, attList), edgeTbl.hasAtts:edgeTblAtts $\Longleftarrow$.

mapping edge-2: (*Construct the source id attribute.*)

c-inst(sourceId, attribute), sourceId.hasName:‘SourceId’, c-inst(‘SourceId’, uldString), sourceId.hasDomain:‘uldString’, edgeTblAtts[1]=sourceId $\Longleftarrow$.

mapping edge-3: (*Construct the tag attribute.*)

c-inst(tag, attribute), tag.hasName:‘Tag’, c-inst(‘Tag’, uldString), tag.hasDomain:‘uldString’, edgeTblAtts[2]=tag $\Longleftarrow$.

mapping edge-4: (*Construct the ordinal attribute.*)
c-inst(ordinal, attribute), ordinal.hasName:‘Ordinal’, c-inst(‘Ordinal’, uldString),
ordinal.hasDomain:‘uldInteger’, edgeTblAtts[3]=ordinal $\Longleftarrow$.

mapping edge-5: (*Construct the target id attribute.*)
c-inst(targetId, attribute), targetid.hasName:‘TargetId’, c-inst(‘TargetId’, uldString),
targetId.hasDomain:‘uldString’, edgeTblAtts[4]=targetId $\Longleftarrow$.

mapping edge-6: (*Construct the data attribute.*)
c-inst(data, attribute), data.hasName:‘Data’, c-inst(‘Data’, uldString),
data.hasDomain:‘uldString’, edgeTblAtts[5]=data $\Longleftarrow$.

mapping edge-7: (*Convert elements to tuples.*)
c-inst(ID, tuple), tuple[1]=S, tuple[2]=T, tuple[3]=O, tuple[4]=R, tuple[5]=D,
c-inst(S, uldString), c-inst(T, uldString), c-inst(O, uldInteger), c-inst(R, uldString),
c-inst(D, uldString), tuple∈edgeTbl $\Longleftarrow$
    findEdge(ID, S, T, O, R, D).

% root node without child element and text content
findEdge(ID, ID, T, 1, ‘NULL’, ‘NULL’) ← c-inst(D, doc), D.hasRoot:E, E.hasTag:T,
    ¬hasSubElemsOrAtts(E), E.hasChildren:C, ¬hasPCData(C), $\mathcal{K}^{E}_{tuple}$(ID).
% root node with text content, but without child element
findEdge(ID, ID, T, 1, ‘NULL’, D) ← c-inst(D, doc), D.hasRoot:E, E.hasTag:T,
    ¬hasSubElemsOrAtts(E), E.hasChildren:C, hasPCData(C, D), $\mathcal{K}^{E}_{tuple}$(ID).
% root node with child element, but without text content
findEdge(ID, ID, T, 1, R, ‘NULL’) ← c-inst(D, doc), D.hasRoot:E, E.hasTag:T,
    hasSubElemsOrAtts(E), $\mathcal{K}^{E}_{children}$(R), E.hasChildren:C, ¬hasPCData(C),
    $\mathcal{K}^{E}_{tuple}$(ID).
% root node with child element and text content
findEdge(ID, ID, T, 1, R,D) ← c-inst(D, doc), D.hasRoot:E, E.hasTag:T,
    hasSubElemsOrAtts(E), $\mathcal{K}^{E}_{children}$(R), E.hasChildren:C, hasPCData(C, D),
    $\mathcal{K}^{E}_{tuple}$(ID).
% attribute
findEdge(ID, S, T, 0, ‘NULL’, D) ← c-inst(E, element), $\mathcal{K}^{E}_{children}$(S), E.hasAtts:AS,
    A∈AS, A.hasName:T, A.hasVal:D, $\mathcal{K}^{A}_{tuple}$(ID)
% element without child element and text content
findEdge(ID, S, T, O, ‘NULL’, ‘NULL’) ← c-inst(E_p, element), E_p.hasChildren:C,

hasChild(C, E, O), $\mathcal{K}^{E_p}_{children}$(S), E.hasName:T, ¬hasSubElemsOrAtts(E),
E.hasChildren:C, ¬hasPCData(C), $\mathcal{K}^{E}_{tuple}$(ID).

% element with text content, but without child element
findEdge(ID, S, T, O, 'NULL', D) ← c-inst(E$_p$, element), E$_p$.hasChildren:C,
hasChild(C, E, O), $\mathcal{K}^{E_p}_{children}$(S), E.hasName:T, ¬hasSubElemsOrAtts(E),
E.hasChildren:C, hasPCData(C, D), $\mathcal{K}^{E}_{tuple}$(ID).

% element with child element, but without text content
findEdge(ID, S, T, O, R, 'NULL') ← c-inst(E$_p$, element), E$_p$.hasChildren:C,
hasChild(C, E, O), $\mathcal{K}^{E_p}_{children}$(S), E.hasName:T, hasSubElemsOrAtts(E), $\mathcal{K}^{E}_{children}$(R),
E.hasChildren:C, ¬hasPCData(C), $\mathcal{K}^{E}_{tuple}$(ID).

% element with child element and text content
findEdge(ID, S, T, O, R, D) ← c-inst(E$_p$, element), E$_p$.hasChildren:C, hasChild(C, E, O),
$\mathcal{K}^{E_p}_{children}$(S), E.hasName:T, hasSubElemsOrAtts(E), $\mathcal{K}^{E}_{children}$(R), E.hasChildren:C,
hasPCData(C, D), $\mathcal{K}^{E}_{tuple}$(ID).

hasPCData(C, D) ← c-inst(C, fullContent), C.hasNode:D, c-inst(D, pcdata).
hasPCData(C, D) ← c-inst(C, fullContent), C.hasContent:C′, hasPCData(C′, D).

hasPCData(C) ← hasPCData(C, D).

hasSubElems(C) ← c-inst(C, fullContent), C.hasNode:N, c-inst(N, element).
hasSubElems(C) ← c-inst(C, fullContent), C.hasChildren:C′, hasSubElems(C′).

hasSubElemsOrAtts(E) ← c-inst(E, element), E.hasAtts:AS, |AS|≠0.
hasSubElemsOrAtts(E) ← c-inst(E, element), E.hasChildren:C, hasSubElems(C).

hasChild(C, E, 1) ← c-inst(C, fullContent), C.hasNode:E, c-inst(E, element).
hasChild(C, E, O) ← c-inst(C, fullContent), C.hasContent:C′, hasChild(C′, E, O′),
O=O′+1.

In contrast to the edge transformation, the "attribute" approach [74, 77] uses schema information to map an XML document to a relational database. The attribute transformation (Transformation 4) is given below. Figure 4.3 shows the result of applying the attribute transformation on the XML DTD and instance of Figure 1.2. The transformation maps each element type to a relation consisting of an id-attribute. Each element-type that is in the content specification of another element type forms a new relation, which stores the sub-element connection between elements. This approach is similar to the mapping of relationship types to relations in the E-R to relational transformation. The attribute *moviedb*

id
id1

movie

id
id2
id3

title

id	*content*
id4	'The Usual Suspects'
id5	'Meet the Parents'

studio

id	*content*
id6	'Gramercy'
id7	'Universal'

genre

id	*content*
id8	'Thriller'
id9	'Comedy'

actor

id	*role*	*content*
id10	'Supporting Actor'	'Spacey, Kevin'
id11	'Leading Actor'	'De Niro, Robert'

moviedbmovie

moviedb	*movie*
id1	id2
id1	id3

movietitle

movie	*title*
id2	id4
id3	id5

moviestudio

movie	*studio*
id2	id6
id3	id7

moviegenre

movie	*genre*
id2	id8
id3	id9

movieactor

movie	*actor*
id2	id10
id3	id11

Figure 4.3: The result of applying the attribute transformation.

approach can also inline sub-elements [14, 74], where based on DTD constraints, an element can be stored directly in its parent element. The transformation below does not consider general inlining, however, we do inline PCDATA where appropriate. Also note that the transformation given below does not consider sub-element ordering as in the edge approach.

The attribute approach only maps elements captured by the XML DTD. However, as in the E-R to relational mapping, we could add rules (and conventions) for storing those XML elements that do not have a corresponding type.

Transformation 4 (*An XML-to-relational attribute transformation*)

mapping att-1: (*Every element type becomes a relation.*)

c-inst(R,relation), R.hasName:N, c-inst(N, uldString), R.hasAtts:AL, c-inst(AL, attList) $\Longleftarrow$

c-inst(T, elemType), T.hasName:N, T.hasAtts:AD, $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{AD}_{attList}$(AL).

mapping att-2: (*Every element type has an identifying attribute in the corresponding relation.*)

c-inst(A, attribute), A∈AL, A.hasName:'id', c-inst('id', uldString), A.hasDomain:'uldString' $\Longleftarrow$

c-inst(T, elemType), T.hasAtts:AD, $\mathcal{K}^{AD}_{attList}$(AL), $\mathcal{K}^{T}_{idatt}$(A), .

mapping att-3: (*Element type attributes become relation attributes.*)

c-inst(A_r, attribute), $A_r \in$AL, A_r.hasName:N, c-inst(N, uldString), A_r.hasDomain:'uldString' $\Longleftarrow$

c-inst(T, elemType), T.hasAtts:AD, $A_e \in$AD, A_e.hasName:N, $\mathcal{K}^{A_e}_{attribute}$($A_r$), $\mathcal{K}^{AD}_{attList}$(AL).

mapping att-4: (*Element types with content have additional relation attributes.*)

c-inst(A, attribute), A$\in$AL, A.hasName:'content', c-inst('content', uldString), A.hasDomain:'uldString' $\Longleftarrow$

c-inst(T, elemType), T.hasModel:CS, c-inst(CS, simpleSpec), CS.isPCData:true, $\mathcal{K}^{T}_{content}$(A), T.hasAtts:AD, $\mathcal{K}^{AD}_{attList}$(AL).

mapping att-5: (*Element-type subelements become edge relations.*)

c-inst(R, relation), R.hasName:N, c-inst(N, uldString), R.hasAtts:AL, c-inst(AL, attList) $\Longleftarrow$

c-inst(T_1, elemType), T_1.hasModel:CS, isSubElement(CS, T_2), T_1.hasName:N_1, T_2.hasName:N_2, N = $N_1 \cdot N_2$, $\mathcal{K}^{T_1,T_2}_{relation}$(R), $\mathcal{K}^{T_1,T_2}_{attList}$(AL).

mapping att-6: (*Subelement edge relations have two identifying attributes.*)

c-inst(A_1, attribute), c-inst(A_2, attribute), AL[1]=A_1, AL[2]=A_2, A_1.hasName:N_1, c-inst(N_1, uldString), A_2.hasName:N_2, c-inst(N_2, uldString) $\Longleftarrow$

c-inst(T_1, elemType), T_1.hasModel:CS, isSubElemType(CS, T_2), T_1.hasName:N_1, T_2.hasName:N_2, $\mathcal{K}^{T_1,T_2}_{relation}$(R), $\mathcal{K}^{T_1,T_2}_{attList}$(AL), $\mathcal{K}^{T_1,T_2}_{att1}$($A_1$), $\mathcal{K}^{T_1,T_2}_{att2}$($A_2$).

mapping att-7: (*Create a table for each element type.*)

c-inst(B, table), d-inst(B, R) $\Longleftarrow$ c-inst(T, elemType), $\mathcal{K}^{T}_{relation}$(R), $\mathcal{K}^{T}_{table}$(B).

mapping att-8: (*Create a table for each subelement type.*)

c-inst(B, table), d-inst(B, R) $\Longleftarrow$

c-inst(T_1, elemType), T_1.hasModel:CS, isSubElemType(CS, T_2), $\mathcal{K}^{T_1,T_2}_{relation}$(R), $\mathcal{K}^{T_1,T_2}_{table}$(B).

mapping att-9: (*Create a tuple for each element.*)

c-inst(P, tuple), P$\in$B $\Longleftarrow$ c-inst(E, element), d-inst(E, T), $\mathcal{K}^{T}_{table}$(B), $\mathcal{K}^{E}_{tuple}$(P).

mapping att-10: (*Create an identifying-attribute value for each element.*)

P[I]=V, c-inst(V, uldString) $\Longleftarrow$

c-inst(E, element), d-inst(E, T), $\mathcal{K}^{T}_{table}$(R), $\mathcal{K}^{T}_{idatt}$(A),
rel/R.hasAtts:AL, *rel*/AL[I]=A, $\mathcal{K}^{E}_{tuple}$(P), $\mathcal{K}^{E}_{idatt}$(V).

mapping att-11: (*Create an attribute value for each element attribute.*)
P[I]=V, c-inst(V, uldString) $\Longleftarrow$
c-inst(E, element), d-inst(E,T), T.hasAtts:AD, $A_e \in$AD, A_e.hasVal:V,
$\mathcal{K}^{A_e}_{attribute}(A_r)$, $\mathcal{K}^{AD}_{attList}$(AL), *rel*/AL[I]=$A_r$, $\mathcal{K}^{E}_{tuple}$(P).

mapping att-12: (*Create an attribute value for elements with only text content.*)
P[I]=V, c-inst(V, uldString) $\Longleftarrow$
c-inst(E, element), d-inst(E, T), T.hasAtts:AD, T.hasModel:CS,
c-inst(CS, simpleSpec), CS.isPCData:true, E.hasChildren:C, C.hasNode:V,
c-inst(V, pcdata), $\mathcal{K}^{T}_{content}$(A), $\mathcal{K}^{AD}_{attList}$(AL), *rel*/AL[I]=A, $\mathcal{K}^{E}_{tuple}$(P).

mapping att-13: (*Create a tuple for each subelement.*)
c-inst(P, tuple), P[1]=V_1, P[2]=V_2, P$\in$B $\Longleftarrow$
c-inst(E_1, element), d-inst(E_1, T_1), E_1.hasChildren:C, isSubElement(C, E_2),
d-inst(E_2, T_2), $\mathcal{K}^{T_1,T_2}_{table}$(B), $\mathcal{K}^{E_1,E_2}_{tuple}$(P), $\mathcal{K}^{E_1}_{idatt}(V_1)$, $\mathcal{K}^{E_2}_{idatt}(V_2)$.

isSubElemType(CS, T) $\leftarrow$ c-inst(CS, mixedSpec), T$\in$CS.
isSubElemType(CS, T) $\leftarrow$ c-inst(CS, elemRE), CS.hasType:T.
isSubElemType(CS, T) $\leftarrow$ c-inst(CS, regExp), CS.hasRE:CS$'$, isSubElemType(CS$'$, T).
isSubElemType(CS, T) $\leftarrow$ c-inst(CS, regExp), CS.hasTail:CS$'$, isSubElemType(CS$'$, T).

isSubElement(C, E) $\leftarrow$ c-inst(C, fullContent), C.hasNode:E, c-inst(E, element).
isSubElement(C, E) $\leftarrow$ c-inst(C, fullContent), C.hasContent:C$'$, isSubElement(C$'$, E).

4.2.3 Schema Transformation

The previous transformations perform data-model-to-data-model conversions. Here, we consider using the ULD transformation language to convert data among particular schemas, where each schema is represented in a different data model.

The following transformation converts the XML source given in Figure 1.2 to a valid RDF source that conforms to the RDF Schema of Figure 1.3. The transformation must consider schematic differences, for example, genre information is represented as data in the XML example and as schema in the RDF example, as well as format differences, for example, names are represented in slightly different ways between the schemas. We make the following assumptions concerning the transformation. The function *toRDFName* takes a name formatted last-name first and returns a new name formatted first-name first. The identifiers *film*, *thriller*, *comedy*, and so on are defined in the RDF Schema. Similarly, the identifiers *movie*, *genre*, and so on are defined in the XML DTD. And finally, the Skolem function $\mathcal{K}^{M}_{id}(\mathrm{U})^{uldURI}$ unifies the variable U with a unique and valid URI value. (Note that *uldURI* is another modifier to the $\mathcal{K}$ function that in general, requests identifiers that conform to a specific atomic type such as strings, integers, URIs, and so on.)

The transformation below is a schema-to-schema mapping for schemas represented in distinct data models. As a result of the mapping, XML data is directly converted to valid RDF data. Alternatively, in schema-mapping approaches, the transformation below must typically be expressed in a common data model. The XML DTD and RDF Schema would first be converted to the common data model (which in general is not trivial). The schema mapping is then defined in the common data model. And finally, the transformation is implemented by converting the mapping rules to an executable program, or by converting source data to the common data model, executing the transformation within the common data model, and then converting the resulting data to the target data model. As shown below, the flexibility of the ULD allows schema mappings to be expressed directly, without the need for additional mappings and conversions to a common data model.

Transformation 5 (*A DTD-to-RDF-Schema transformation*)

mapping schema-1: (*Movie elements become film resources.*)

c-inst(R, simpleRes), d-inst(R, film), R.hasURI:U, c-inst(U, uldURI) $\Longleftarrow$
    d-inst(M, movie), $\mathcal{K}^{M}_{resource}(\mathrm{R})$, $\mathcal{K}^{M}_{id}(\mathrm{U})^{uldURI}$.

mapping schema-2: (*Thriller movies become thriller-film resources.*)

d-inst(R, thriller) $\Longleftarrow$
    d-inst(M, movie), M.hasChildren:C, isSubElement(C, M′), d-inst(M′,genre),
    M′.hasChildren:C′, hasPCData(C′, ‘Thriller’), $\mathcal{K}^{M}_{resource}(\mathrm{R})$.

mapping schema-3: (*Comedy movies become thriller-film resources.*)

d-inst(R, comedy) $\Longleftarrow$
    d-inst(M, movie), M.hasChildren:C, isSubElement(C, M′), d-inst(M′,genre),
    M′.hasChildren:C′, hasPCData(C′, ‘Comedy’), $\mathcal{K}^{M}_{resource}(\mathrm{R})$.

mapping schema-4: (*Title elements become film title triples.*)

c-inst(V, literal), c-inst(T, triple), T.hasPred:title, T.hasSubj:R, T.hasObj:V $\Longleftarrow$
d-inst(M, movie), M.hasChildren:C, isSubElement(C, M′), d-inst(M′,title), M′.hasChildren:C′, hasPCData(C′, V), $\mathcal{K}^{M}_{resource}$(R), $\mathcal{K}^{M}_{title}$(T).

mapping schema-5: (*Actor elements become actor resources.*)

c-inst(R, simpleRes), d-inst(R, actor), R.hasURI:U, c-inst(U, uldURI) $\Longleftarrow$
d-inst(A, actor), $\mathcal{K}^{A}_{resource}$(R), $\mathcal{K}^{A}_{id}$(U)^{uldURI}.

mapping schema-6: (*Actor names become hasName properties.*)

c-inst(V, literal), c-inst(T, triple), T.hasPred:hasName, T.hasSubj:R, T.hasObj:V $\Longleftarrow$
d-inst(A, actor), A.hasChildren:C, hasPCData(C, N), V=*toRDFName*(N), $\mathcal{K}^{A}_{resource}$(R), $\mathcal{K}^{A}_{name}$(T).

mapping schema-7: (*Actor subelements are connected to film resources.*)

c-inst(T, triple), T.hasPred:involved, T.hasSubj:R$_m$, T.hasObj:R$_a$ $\Longleftarrow$
d-inst(M, movie), M.hasChildren:C, isSubElement(C, A), d-inst(A, actor), $\mathcal{K}^{M}_{resource}$(R$_m$), $\mathcal{K}^{A}_{resource}$(R$_a$), $\mathcal{K}^{M,A}_{involved}$(T).

isSubElement(C, E) $\leftarrow$ c-inst(C, fullContent), C.hasNode:E, c-inst(E, element).
isSubElement(C, E) $\leftarrow$ c-inst(C, fullContent), C.hasContent:C′, isSubElement(C′, E).

hasPCData(C, D) $\leftarrow$ c-inst(C, fullContent), C.hasNode:D, c-inst(D, pcdata).
hasPCData(C, D) $\leftarrow$ c-inst(C, fullContent), C.hasContent:C′, hasPCData(C′, D).

4.2.4 Transformations for Standard Serializations

The last transformation (Transformation 6) of this section converts RDF data directly into XML data. Figure 4.4 gives the result of applying the mapping to the RDF configuration shown in Figure 1.3. The resulting XML document is similar to the standard RDF serialization in XML [47]. A serialization is, fundamentally, a data-model mapping in which one data model is used as the interchange format for information expressed in another data model. One of the most widely used data models for serialization is XML. For example, both RDF and Topic Maps have multiple XML serializations.

```
<rdf>
   <description about="#m1">
      <title>The Usual Suspects</title>
      <involved resource="#ks"/>
      <won resource="#a1"/>
   </description>
   <description about="#ks">
      <hasName>Kevin Space</hasName>
   </description>
   <description about="#a1">
      <recipient resource="#ks"/>
      <category>Supporting Actor</category>
   </description>
</rdf>
```

Figure 4.4: The result of applying the RDF serialization mapping.

We note that it is not possible to define an XML DTD for the standard RDF serialization. In particular, the element tags present in an XML document depend on the property names used in the RDF source. Therefore, one benefit of using the ULD transformation language is that it provides a formal, explicit, and executable definition of the serialization. In addition, serializations often use a number of conventions and mix data and schema levels. For example, RDF schema information and instance data are both mapped to XML elements and attributes (that is, instance data) in the standard RDF serialization. The flexibility of the ULD enables the specification of these various styles of mappings that mix data-model, schema, and data levels.

Transformation 6 (*An RDF-to-XML serialization*)

mapping rdfxml-1: (*Generate a document and an RDF root tag.*)
c-inst(rdfDoc, doc), c-inst(rdfRoot, element), rdfRoot.hasTag:'rdf',
rdfRoot.hasAtts:rdfRootAtts, c-inst('rdf', uldString),
c-inst(rdfRootAtts, emptyContent) $\Longleftarrow$.

mapping rdfxml-2: (*Each resource that is the subject of a triple generates an element.*)
c-inst(E, element), E.hasTag:'description', c-inst('description', uldString), E.hasAtts:AS,
c-inst(AS, attSet) $\Longleftarrow$
c-inst(T, triple), T.hasSubj:S, $\mathcal{K}^S_{desc}$(E), $\mathcal{K}^S_{attList}$(AS).

mapping rdfxml-3: (*Add first subelement of the root.*)

rdfRoot.hasChildren:rdfRootContent, c-inst(rdfRootContent, fullContent), rdfRootContent.hasNode:E $\Longleftarrow$

c-inst(T, triple), T.hasSubj:S, $\mathcal{K}^{S}_{desc}$(E), *xml*/$\neg$c-inst(rdfRootContent, fullContent).

mapping rdfxml-4: (*Add remaining subelements of the root.*)

C.hasContent:C$'$, c-inst(C$'$, fullContent), C$'$.hasNode:E $\Longleftarrow$

c-inst(T, triple), T.hasSubj:S, $\mathcal{K}^{S}_{desc}$(E), findLastContent(rdfRootContent, C), $\mathcal{K}^{S}_{content}$(C$'$).

mapping rdfxml-5: (*If every subject has been converted, add empty content.*)

C.hasContent:C$'$, c-inst(C$'$, emptyContent) $\Longleftarrow$

$\neg$existsSubjNotConverted(rdfRootContent), findLastContent(rdfRootContent, C), $\mathcal{K}_{emptycont}$(C$'$).

mapping rdfxml-6: (*Add the subject resource URI as an attribute.*)

E.hasAtts:AS, c-inst(AS, attSet), A$\in$AS, c-inst(A, attribute), A.hasName:'about', c-inst('about', uldString), A.hasVal:U, c-inst(U, cdata) $\Longleftarrow$

c-inst(T, triple), T.hasSubj:S, S.hasURI:U, $\mathcal{K}^{S}_{desc}$(E), $\mathcal{K}^{S}_{attSet}$(AS), $\mathcal{K}^{S}_{attribute}$(A).

mapping rdfxml-7: (*Generate first subelement of subject element.*)

E.hasChildren:C, c-inst(C, fullContent), C.hasNode:E$'$, c-inst(E$'$, element), E$'$.hasTag:U, E$'$.hasAtts:AS, c-inst(AS, attSet), E$'$.hasContent:C$_p$, c-inst(C$_p$, fullContent), C$_p$.hasNode:O, c-inst(O, pcdata), C$_p$.hasContent:C$_e$, c-inst(C$_e$, emptyContent) $\Longleftarrow$

c-inst(T, triple), T.hasSubj:S, T.hasPred:P, P.hasURI:U, T.hasObj:O, c-inst(O, literal), $\mathcal{K}^{S}_{desc}$(E), $\mathcal{K}^{S}_{children}$(C), *xml*/$\neg$c-inst(C, fullContent), $\mathcal{K}^{S,P,O}_{element}$(E$'$), $\mathcal{K}^{S,P,O}_{predicate}$(C$_p$), $\mathcal{K}^{S,P,O}_{empty}$(C$_e$), $\mathcal{K}^{S,P,O}_{attSet}$(AS).

mapping rdfxml-8: (*Generate first subelement of subject element.*)

E.hasChildren:C, c-inst(C, fullContent), C.hasNode:E$'$, c-inst(E$'$, element), E$'$.hasTag:U$_p$, E$'$.hasAtts:AS, c-inst(AS, attSet), A$\in$AS, c-inst(A, attribute), A.hasName:'resource', c-inst('resource', uldString), A.hasVal:U$_o$, c-inst(U$_o$, cdata), E$'$.hasContent:C$'$, c-inst(C$'$, emptyContent) $\Longleftarrow$

c-inst(T, triple), T.hasSubj:S, T.hasPred:P, P.hasURI:U$_p$, T.hasObj:O, O.hasURI:U$_o$, $\mathcal{K}^{S}_{desc}$(E), $\mathcal{K}^{S}_{children}$(C), *xml*/$\neg$c-inst(C, fullContent), $\mathcal{K}^{S,P,O}_{element}$(E$'$), $\mathcal{K}^{S,P,O}_{empty}$(C$'$), $\mathcal{K}^{S,P,O}_{attSet}$(AS), $\mathcal{K}^{S,P,O}_{object}$(A).

mapping rdfxml-9: (*Generate remaining subelements of subject element.*)

C.hasContent:C$'$, c-inst(C$'$, fullContent), C$'$.hasNode:E$'$, c-inst(E$'$, element), E$'$.hasTag:U,

E′.hasAtts:AS, c-inst(AS, attSet), E′.hasContent:C$_p$, c-inst(C$_p$, fullContent), C$_p$.hasNode:O, c-inst(O, pcdata), C$_p$.hasContent:C$_e$, c-inst(C$_e$, emptyContent) $\Longleftarrow$
c-inst(T, triple), T.hasSubj:S, T.hasPred:P, P.hasURI:U, T.hasObj:O, c-inst(O, literal), $\mathcal{K}^{S}_{desc}$(E), $\mathcal{K}^{S}_{children}$(C), findLastContent(C, C′), $\mathcal{K}^{S,P,O}_{element}$(E′), $\mathcal{K}^{S,P,O}_{predicate}$(C$_p$), $\mathcal{K}^{S,P,O}_{empty}$(C$_e$), $\mathcal{K}^{S,P,O}_{attSet}$(AS).

mapping rdfxml-10: (*Generate remaining subelements of subject element.*)
C.hasContent:C′, c-inst(C′, fullContent), C′.hasNode:E′, c-inst(E′, element), E′.hasTag:U$_p$, E′.hasAtts:AS, c-inst(AS, attSet), A∈AS, c-inst(A, attribute), A.hasName:‘resource’, c-inst(‘resource’, uldString), A.hasVal:U$_o$, c-inst(U$_o$, cdata), E′.hasContent:C′, c-inst(C′, emptyContent) $\Longleftarrow$
c-inst(T, triple), T.hasSubj:S, T.hasPred:P, P.hasURI:U$_p$, T.hasObj:O, O.hasURI:U$_o$, $\mathcal{K}^{S}_{desc}$(E), $\mathcal{K}^{S}_{children}$(C), findLastContent(C,C′), $\mathcal{K}^{S,P,O}_{element}$(E′), $\mathcal{K}^{S,P,O}_{empty}$(C′), $\mathcal{K}^{S,P,O}_{attSet}$(AS), $\mathcal{K}^{S,P,O}_{object}$(A).

mapping rdfxml-11: (*If every triple has been converted, add empty content.*)
C′.hasContent:C$_e$, c-inst(C$_e$, emptyContent) $\Longleftarrow$
c-inst(T, triple), T.hasSubj:S, ¬existsTripleNotConverted(S), $\mathcal{K}^{S}_{desc}$(E), *xml*/E.hasContent:C, findLastContent(C, C′), $\mathcal{K}^{S}_{emptycont}$(C$_e$).

existsSubjNotConverted(C) ← *xml*/c-inst(C, fullContent), c-inst(T, triple), T.hasSubj:S, $\mathcal{K}^{S}_{desc}$(E), ¬isSubElement(C, E).

existsTripleNotConverted(S) ← c-inst(T, triple), T.hasSubj:S, $\mathcal{K}^{S}_{desc}$(E), E.hasContent:C, T.hasPred:P, T.hasObj:O, $\mathcal{K}^{S,P,O}_{e}lement$(E′), ¬isSubElement(C, E′).

isSubElement(C, E) ← *xml*/c-inst(C, fullContent), *xml*/C.hasNode:E, *xml*/c-inst(E, element).
isSubElement(C, E) ← *xml*/c-inst(C, fullContent), *xml*/C.hasContent:C′, *xml*/isSubElement(C′, E).

findLastContent(C$_1$, C$_1$) ← *xml*/c-inst(C$_1$, fullContent), ¬hasFilledContent(C$_1$).
findLastContent(C$_1$, C$_3$) ← *xml*/c-inst(C$_1$, fullContent), *xml*/C$_1$.hasContent:C$_2$, *xml*/¬ c-inst(C$_2$, emptyContent), findLastContent(C$_2$, C$_3$).

hasFilledContent(C) ← *xml*/c-inst(C, fullContent), *xml*/C.hasContent:C′.

4.3 Analyzing ULD Transformations

We consider two applications of reasoning for ULD mapping rules. By reasoning, we mean analyzing mapping rules prior to their execution using ULD axioms and data-model constraints. The first application we consider ensures that a mapping rule is *correct*, that is, the resulting target configuration does not violate the ULD axioms and data-model constraints. The second application of reasoning is directed at helping a user define a transformation. In particular, we want to exploit the ULD axioms and data-model constraints to "fill-out" incomplete mapping rules. As an ambitious example, a user may specify that E-R entity types should be mapped to relations in the relational model without giving the details of such a mapping, where the rest of the mapping is filled in by a reasoning system. (Note that similar approaches exist for schema mappings [68] and we consider only simple cases of the more general problem in this chapter.)

This section focuses on ensuring mapping rules are correct with respect to ULD axioms and briefly discusses the use of data-model constraints for both ensuring correctness and elaborating partial mapping rules.

4.3.1 Well-Typed Transformations

Here we describe a structural approach for verifying that a transformation is *well-typed*. Based on the axioms of the ULD (from Chapter 2), we define a set of rules to annotate variables used in mapping rules with type information, and then use this type information to verify that a transformation is well-formed (with respect to the types).

Our overriding goal is to ensure that a transformation will result in a consistent target configuration and notify the user if the transformation can produce an inconsistent result (or in general, if a mapping rule is not well-formed). We only consider individual mapping rules. In general, a transformation should be considered as a whole; each individual mapping rule can be consistent, but when taken together, they may not produce a consistent transformation.[4]

[4]Note that the converse is not true, that is, if a mapping rule can produce an inconsistent configuration, it is not possible to construct another mapping rule to make the configuration consistent. Intuitively, a configuration is inconsistent if it asserts a fact that violates a ULD or data-model constraint. Therefore,

A mapping rule is inconsistent if it can produce target facts that violate ULD axioms or data-model constraints. We also consider whether the body of a mapping rule can be satisfied, that is, whether it is possible to construct a valid configuration that produces a variable assignment for the rule (or for rule bodies that do not contain variables, whether they are false for every possible configuration). No updates will result from executing a mapping rule with a rule body that is not satisfiable. This mapping rule is not considered well-formed even though it does not produce an inconsistent target configuration.

Checking for well-typed mapping rules does not guarantee that a mapping rule produces a consistent target configuration, only that the result is consistent with respect to typing constraints. For example, consider the following mapping rule (Example 11), which maps E-R attributes to relations in the relational data model.

Example 11 *(A well-typed transformation rule)*

c-inst(T, relation), T.hasName:N, c-inst(N, uldString) $\Longleftarrow$
    c-inst(A, attribute), A.hasName:N, $\mathcal{K}^A$(T).

This rule is considered well-typed (which we define below) and thus is valid with respect to typing. However, the target configuration generated from the rule may not be consistent. In particular, multiple attributes with the same name are allowed in the source configuration (which uses an E-R data model), however, each relation in the target configuration (a relational data model) must contain a unique name. Therefore, if there is more than one attribute with the same name in the source, the resulting target configuration will be inconsistent with respect to the relational data-model constraint (hasName keyOf relation).

The *type annotation* rules in Figures 4.5, 4.6, and 4.7 are used to determine the types of variables and constants in a mapping rule. If the antecedent of a rule holds (the expression above the line), the type annotation in the consequent of the rule is asserted (the expression below the line). A type annotation takes the form $c/v :: t$, which states that v, a variable

such a fact would need to be removed from the configuration to make it consistent, and it is not possible to remove facts from a configuration using the ULD transformation language.

or constant in the mapping rule, has type t in configuration c. An expression such as c/c-inst(x, p) (for x and p constants or variables and c a configuration name) in an annotation rule is matched against a similar expression in a mapping rule. (Note that we assume the default source and target configurations if the configuration name is not given, as in the previous example.) For example, the formula *rel*/c-inst(T, relation) in the previous mapping rule (Example 11) would match the expression c/c-inst(x, p) in a type annotation rule, where x is unified with T, p is unified with the constant 'relation', and c is unified with the (default) configuration *rel*.

Type annotation rules may also use data-model information from a configuration to annotate variables and constants. Data-model information is accessed using expressions of the form $c \models b$, which are true if the variables of a formula b can be unified in configuration c. The expression $c \models b$ is essentially a ULD query, which finds a variable assignment for b in configuration c through unification. Data-model expressions can require certain terms to be ground prior to their application. For example, the expression $c \models$ ct-inst(p^*, s) requires p to be either a constant, or a variable unified with a constant.

Type annotation rules are repeatedly applied to each formula in a mapping rule until no additional types are generated. That is, annotation rules are applied until a fixpoint is reached. We note that this approach to typing is similar to the system of Mycroft and O'Keefe [58], which was developed specifically for logic programming in Prolog. We note that the Mycroft and O'Keefe type system is considerably more complex, for example, formulas are annotated with types in addition to variables and constants.

We permit variables and constants to be annotated with construct-type and construct identifiers. That is, the set of permissible types for a source or target formula in a mapping rule consists of the ULD construct types and the constructs defined in the source or target configuration, respectively. Intuitively, if a variable in a mapping rule ranges over constructs, the variable is annotated with the appropriate construct type. Similarly, if a variable in a mapping rule ranges over construct instances, the variable is annotated with the appropriate construct. (Note that as an extension, we could also consider annotating construct instances with other construct instances according to conformance and *d-inst* relationships.)

$$(1)\ \frac{c/\text{ct-inst}(p,\ s)}{c/p :: s}$$

$$(2)\ \frac{c/\text{c-inst}(x,\ p)}{c/x :: p}$$

$$(3)\ \frac{c/\text{c-inst}(x,\ p),\ c \models \text{ct-inst}(p^*,\ s)}{c/p :: s}$$

$$(4)\ \frac{c/x.s{:}y,\ c/x :: p,\ c \models p^*.s^*\texttt{->}q}{c/y :: q}$$

$$(5)\ \frac{c/\text{d-inst}(x,\ y),\ c/x :: p,\ c \models \text{conf}(p^*,\ q,\ d,\ r)}{c/y :: q}$$

$$(6)\ \frac{c/y :: q,\ c \models \text{unionof}(p,\ q^*)}{c/y :: p}$$

Figure 4.5: The basic type annotation rules for mappings.

$$(7)\ \frac{c/y \in x,\ c/x :: p,\ c \models \text{setof}(p^*,\ q)}{c/y :: q}$$

$$(8)\ \frac{c/y \in x,\ c/x :: p,\ c \models \text{listof}(p^*,\ q)}{c/y :: q}$$

$$(9)\ \frac{c/y \in x,\ c/x :: p,\ c \models \text{bagof}(p^*,\ q)}{c/y :: q}$$

$$(10)\ \frac{c/x[i] = y,\ c/x :: p,\ c \models \text{listof}(p^*,\ q)}{c/y :: q}$$

$$(11)\ \frac{c/y{\in}{\in}x{=}l,\ c/x :: p,\ c \models \text{bagof}(p^*,\ q)}{c/y :: q}$$

Figure 4.6: The collection type annotation rules for mappings.

$$(12)\ \frac{c/x[i] = y}{c/i :: \text{uldInteger}} \qquad (13)\ \frac{c/|x| = l}{c/l :: \text{uldInteger}}$$

$$(14)\ \frac{c/y{\in}{\in}x = l}{c/l :: \text{uldInteger}} \qquad (15)\ \frac{c/x.s{:}y}{c/s :: \text{uldString}}$$

$$(16)\ \frac{c/p.s\text{->}q}{c/s :: \text{uldString}} \qquad (17)\ \frac{c/\text{conf}(p,\ q,\ d,\ r)}{c/d :: \text{uldString}}$$

$$(18)\ \frac{c/\text{conf}(p,\ q,\ d,\ r)}{c/r :: \text{uldString}} \qquad (19)\ \frac{\mathcal{K}_l^{v_1,\ldots,v_n}(v)^t}{v :: t}$$

Figure 4.7: The selector, conformance-cardinality, and identifier-generation type annotation rules for mappings.

The first type annotation rule states that, if p (a constant or variable) is a *ct-inst* of s (expressed as a formula in a mapping rule), the type of p is s. The second rule states that if x is a *c-inst* of p, the type of x is p. The third rule accesses data-model information for the case when a *ct-inst* relation is not explicitly given in the mapping rule. To illustrate, the following annotations result from applying the second and third rules to the previous mapping (Example 11).

er/A :: attribute
er/attribute :: struct-ct
er/uldString :: atomic-ct
rel/T :: relation
rel/N :: uldString
rel/relation :: struct-ct
rel/uldString :: atomic-ct

The fourth typing rule follows from axiom 27 (of Chapter 2) and states that the type of a component-selector value is the type of the corresponding component-definition value. To apply this annotation rule to the previous mapping, we must obtain the component definition in the corresponding source configuration. Applying the rule to the mapping in Example 11 adds the type annotation *er*/N :: uldString.

The fifth annotation rule follows from axiom 2 and 22 (as discussed in Chapter 2). Namely, we can use the conformance relationship in a configuration to determine the type of the construct instance serving as schema in a *d-inst* relationship. The sixth annotation rule enumerates union types. Rules seven through eleven assign collection types to members. Rules twelve through fourteen associate the index of a list, length of a collection, and number of repeated bag elements, respectively, to an integer atomic type. Rules fifteen and sixteen assign an atomic type to selectors; and rules seventeen and eighteen assign atomic types to conformance cardinality constraints. Finally, rule nineteen associates an atomic type (such as *uldURI*) to a generated identifier, if such an atomic type is given. Note that we do not specify a particular configuration for the generated identifier's type. Instead, we treat the type annotation as being applicable in any configuration in which the variable is used.

Once type annotations for a mapping rule are determined, we use the typing rules in Figure 4.8 to check for well-typed mappings. The typing rules are expressed using first-order logic and differ from the annotation rules in that they define constraints that must be satisfied for the mapping to be well-typed. The typing rules use notation similar to annotation rules, namely, the expression c/b is matched against formulas in the mapping, and the expression $c \models b$ is matched against data-model information.

The first five typing rules require component selectors, *d-inst* expressions, membership expressions, list index expressions, and bag membership expressions, respectively, to be well-typed. The sixth rule requires every variable or constant with multiple types to have at least one non-union type. The seventh rule requires at most one such non-union type for each variable or constant. The eighth rule requires each type for variables or constants with more than one type to be related through a union definition. Note that the *unionConnected* constraint is defined by axiom 57 of Chapter 2. The ninth rule states that every variable and constant in a mapping rule (with a type annotation) has a ground type (that is, the type is either a variable with a known assignment or a constant). We assume the *ground* function returns true if its argument is ground, and false otherwise. Finally, the last typing rule restricts variables and constants with more than one distinct *atomic-ct* type (regardless of the configuration) to be type compatible. For example,

(i) $\forall(c,x,s,y)\ (c/x.s{:}y) \rightarrow \exists(p,q)\ (c/x :: p) \wedge (c/y :: q) \wedge (c/s ::$ uldString$) \wedge$
$(c \models p.s\text{->}q)$

(ii) $\forall(c,x,y)\ (c/\text{d-inst}(x,\ y)) \rightarrow \exists(p,q,d,r)\ (c/x :: p) \wedge$
$(c \models \text{conf}(\ p,\ q,\ d,\ r)) \wedge (c/y :: q) \wedge (c/d ::$ uldString$) \wedge (c/r ::$ uldString$)$

(iii) $\forall(c,x,y)\ (c/y{\in}x) \rightarrow \exists(p,q)\ (c/x :: p) \wedge$
$[c \models (\text{setof}(p,\ q) \vee \text{listof}(p,\ q) \vee \text{bagof}(p,\ q))] \wedge (c/y :: q)$

(iv) $\forall(c,x,y,i)\ (c/x[i]{=}y) \rightarrow \exists(p,q)\ (c/x :: p) \wedge (c \models \text{listof}(p,\ q)) \wedge (c/y :: q) \wedge$
$(c/i ::$ uldInteger$)$

(v) $\forall(c,x,y,l)\ (c/y{\in\in}x{=}l) \rightarrow \exists(p,q)\ (c/x :: p) \wedge (c \models \text{bagof}(p,\ q)) \wedge (c/y :: q) \wedge$
$(c/l ::$ uldInteger$)$

(vi) $\forall(c,v,t_1,...,t_n)\ (c/v :: t_1) \wedge (c/v :: t_2) \wedge ...\ (c/v :: t_n) \rightarrow \exists(i)\ (1 \leq i \leq n) \wedge$
$(c \models \neg\text{ct-inst}(t_i,\ \text{union-ct}))$

(vii) $\forall(c,v,t_1,t_2)\ (c/v :: t_1) \wedge (c/v :: t_2) \wedge (c \models \neg\text{ct-inst}(t_1,\ \text{union-ct})) \wedge$
$(c \models \neg\text{ct-inst}(t_2,\ \text{union-ct})) \rightarrow (t_1{=}t_2)$

(viii) $\forall(c,v,t_1,t_2)\ (c/v :: t_1) \wedge (c/v :: t_2) \wedge (t_1 \neq t_2) \rightarrow \textit{unionConnected}(t_1,\ t_2)$

(ix) $\forall(c,v,t)\ (c/v :: t) \rightarrow \textit{ground}(t)$

(x) $\forall(c_1,c_2,v,t_1,t_2)\ (c_1/v :: t_1) \wedge (c_2/v :: t_2) \wedge (c_1 \models \text{ct-inst}(t_1,\ \text{atomic-ct})) \wedge$
$(c_2 \models \text{ct-inst}(t_2,\ \text{atomic-ct})) \rightarrow \textit{atomicCompatible}(t_1,t_2)$

Figure 4.8: The typing rules for ULD mappings.

`pcdata` and `uldString` are compatible atomic types whereas `uldInteger` and `uldString` are not. Note that, in the ULD, we do not provide an explicit mechanism for defining atomic-type compatibility, however, we assume that compatibility can be computed, for example, based on an external description.

To illustrate the use of typing rules, consider the following mapping rule that attempts to map E-R entities to relations in the relational data model. Using the type annotation rules, the variable AS is annotated with the construct `attSet`, which is defined as a *set-ct* construct in the E-R data model. However, the formula AS.hasName:N in the body of the rule treats AS as though it were a *struct-ct* construct. This formula violates the first typing rule in Figure 4.8, which states that a formula $x.s{:}y$ is valid if the type of x has a component-selector definition for s. Here, since `attSet` is a set construct, the type for AS cannot have a component-selector definition.

c-inst(T, relation), T.hasName:N $\Longleftarrow$

c-inst(E, entity), d-inst(E, T), T.hasAtts:AS, AS.hasName:N.

Similarly, the following mapping rule is also not well-typed. Unlike the previous example, the following mapping may result in an inconsistent target configuration, because the variable T in the head of the mapping rule is annotated with the two types, relation and attList, neither of which are union constructs (violating the seventh typing rule in Figure 4.8.)

c-inst(T, relation), T.hasAtts:AS, c-inst(T, attList) $\Longleftarrow$
c-inst(E, entity), d-inst(E, T), T.hasAtts:AS.

Finally, while not discussed here, the annotation and typing rules can also be applied directly to mapping rules that use an intensional formula. For mapping rules that use intensional formulas, we can determine type annotations by recursively applying the annotation rules to the variables and constants of the Datalog rules that define the intensional predicate. In other words, the type annotation rules can be directly applied to the Datalog rules that construct intensional predicates in a transformation.

4.3.2 Mapping Rules and Data-Model Constraints

Constraint macros offer a class of data-model constraints that can be used directly to determine whether a mapping rule is satisfiable with respect to the available constraint macros. Because there are a finite number of constraint macros, we can define specific procedures for each macro to check whether a mapping violates a target configuration.

Here we consider configuration-based uniqueness constraints, which take the form (p_1, p_2, ..., p_n keyOf c), where $n \geq 1$ and c is a *struct-ct* construct (or possibly a *union-ct* construct). Given a mapping rule $h \Longleftarrow b$, where h and b each consist of one or more formulas, executing the rule may result in an inconsistent target configuration if each of the following conditions hold.

1. A set of component-selector formulas are present in the head of the rule, where each target formula originates from the same variable or constant x in configuration f_t, and the selectors represent a configuration-based uniqueness constraint. That is,

$\{f_t/x.p_{t1}{:}v_{t1},\ f_t/x.p_{t2}{:}v_{t2},\ ...,\ f_t/x.p_{tn}{:}v_{tn}\} \subseteq h$, $f_t/x :: c_t$, and (p_{t1}, p_{t2}, ..., p_{tn} keyOf c_t) is a constraint in f_t.

2. A set of component-selector formulas are present in the body of the rule, where each formula originates from the same variable or constant y in configuration f_s, such that y is mapped to x and each selector value of y is mapped to a corresponding selector value for x. That is, $\{f_s/y.p_{s1}{:}v_{s1},\ f_s/y.p_{s2}{:}v_{s2},\ ...,\ f_s/y.p_{sn}{:}v_{sn}\} \subseteq b$, and $y{=}x$ or $\mathcal{K}^y_l(x)$, $v_{s1}{=}v_{t1}$ or $\mathcal{K}^{v_{s1}}_{l_1}(v_{t1})$, $v_{s2}{=}v_{t2}$ or $\mathcal{K}^{v_{s2}}_{l_2}(v_{t2})$, ..., and $v_{sn}{=}v_{tn}$ or $\mathcal{K}^{v_{sn}}_{l_n}(v_{tn})$.

3. The type of y in (2) does not have a corresponding configuration-based uniqueness constraint. That is, $f_s/y :: c_s$, and (p_1, p_2, ..., p_m keyOf c_s) is not a constraint in f_s such that $\{p_{s1}, p_{s2}, ..., p_{sn}\} \subseteq \{p_1, p_2, ..., p_m\}$.

To illustrate the use of these conditions, consider the following mapping rule (taken from Example 11).

c-inst(T, relation), T.hasName:N, c-inst(N, uldString) $\Longleftarrow$
    c-inst(A, attribute), A.hasName:N, $\mathcal{K}^A$(T).

As previously mentioned, the mapping rule may result in an inconsistent target configuration because relation names are required to be unique, whereas attribute names are not. The following shows that each of the previous conditions hold for the rule, and thus, the mapping may result in an inconsistent target configuration. (Note that the rule results in an inconsistent target configuration only if there is more than one attribute in the source.)

1. $\{rel/$T.hasName:N$\} \subseteq h$, $rel/$T :: relation, and (hasName keyOf relation) is a constraint in rel.

2. $\{er/$A.hasName:N$\} \subseteq b$, $\mathcal{K}^A$(T), and N=N.

3. $rel/$A :: attribute and (hasName keyOf attribute) is not a constraint in er.

We note that checking constraint satisfiability for mapping rules requires (1) propagating constraints surrounding formulas in the body of the rule to formulas in the head of the rule, and (2) ensuring that the propagated constraints do not violate the constraints associated with the target data models. For arbitrary ULD data-model constraints expressed using stratified Datalog¬, checking constraint satisfiability requires checking query satisfiability [50, 67] (that is, checking that the target query, the propagated constraints, and the existing constraints are satisfiable), which in general is undecidable [50, 51]. However, a number of useful constraints can still be exploited, such as the uniqueness constraint shown here.

4.3.3 Cardinality Constraints on Conformance

Similar to constraint macros, we can use cardinality constraints on conformance relationships to determine whether a mapping rule can generate an inconsistent target configuration. Here, we briefly sketch the approach. Consider the following mapping rule (changed slightly from er-rel-8 to make it inconsistent).

$$\begin{aligned}&\text{c-inst(P, tuple), c-inst(B, table), P} \in \text{B, d-inst(P, R)} \Longleftarrow \\ &\qquad \text{c-inst(E, entity), d-inst(E, T)}, \mathcal{K}^{E}_{tuple}(\text{P}), \mathcal{K}^{T}_{table}(\text{B}), \mathcal{K}^{T}_{relation}(\text{R}).\end{aligned}$$

As shown, the rule generates a relational tuple for every entity in an E-R configuration. This mapping rule can result in an inconsistent target configuration. In particular, the E-R data model permits entities to be instances of multiple entity types, whereas the relational model requires a tuple to be an instance of exactly one type.

The following type annotations are generated for the mapping rule.

er/E :: entity
er/T :: entityType
rel/P :: tuple
rel/R :: relation
rel/B :: table

In addition, the formula d-inst(E, T) in the body of the rule and the formula d-inst(P, R) in the head of the rule assert that E is a data instance of T, and P is a data instance of R. We also know from the formulas $\mathcal{K}^{E}_{tuple}$(P) and $\mathcal{K}^{T}_{relation}$(R) that E generates the identifier P, and T generates the identifier R. Therefore, the cardinality restriction on the conformance relationship between entities and entity types must not violate the cardinality restriction on the conformance relationship between tuples and relations.

The following definitions are taken from the data-model descriptions in Chapter 3 and state the conformance cardinality constraints for E-R entities and relational tuples. As shown, tuples must be data instances of exactly one relation, however, entities can be data instances of any number of entity types. Thus, the conformance constraint in the head of the mapping rule may be violated by the conformance constraints in the body of the rule.

```
construct tuple = list of uldValue conf(domain=*,range=1):relation
construct entity = {hasProps->propSet} conf(domain=*,range=*):entityType
```

In this case, the cardinality restriction in the head of the rule forms a specific uniqueness constraint that may be violated by the body of the rule. The following mapping, however, is satisfiable with respect to cardinality constraints on conformance. This mapping is exactly er-rel-8.

$$\begin{aligned}&\text{c-inst(P, tuple), c-inst(B, table), P}\in\text{B, d-inst(P, R)} \Longleftarrow \\ &\qquad \text{c-inst(E, entity), d-inst(E, T), } \mathcal{K}^{E,T}_{tuple}(\text{P}),\ \mathcal{K}^{T}_{table}(\text{B}),\ \mathcal{K}^{T}_{relation}(\text{R}).\end{aligned}$$

4.3.4 Inference and Transformation

We end this chapter by giving a simple example of automatically filling-in partial mapping rules using ULD axioms. We then briefly outline how a general approach might proceed when additional constraints are considered.

The following (Example 12) is a simple example of a partial mapping rule. In particular, the mapping rule does not specify what the variable N is an instance of. The rule is well-typed, but can result in an inconsistent target configuration because every identifier (in this case N), other than a construct type, must be defined as a construct or a construct instance. In this case, the formula c-inst(N, uldString) should be added to the head of the rule.

Example 12 (*A partial mapping rule*)

$$\text{c-inst(R, relation), R.hasName:N} \Longleftarrow \text{c-inst(T, entityType), T.hasName:N.}$$

We use the following *elaboration* rules to fill-in *c-inst* relationships in partial mappings. The rules are similar to the type annotation rules of Figure 4.8 in that they are applied directly to mapping rules. However, instead of annotating constants and variables with types, an elaboration rule adds a *c-inst* formula to the head of a mapping rule. An elaboration rule takes the form **if** h **then** c/c-inst(x,p), where h is a list of formulas and c is a target configuration. If the set of formulas h can be unified with the head of a mapping rule and data-model information from a configuration (rule eleven is a special case that we describe below), the formula c/c-inst(x,p) is added to the head of the mapping rule if it is not already present. Note that the elaboration rules below rely on type annotations and data-model information.

(1). **if** $c/x.s{:}y$, $c/x :: p$, $c \models p^*.s^*\text{->}q$ **then** c/c-inst(y, q)

(2). **if** c/d-inst(x,y), $c/x :: p$, $c \models$ conf(p^*, q, d, r) **then** c/c-inst(y, q)

(3). **if** $c/y :: q$, $c \models$ unionof(p, q^*) **then** c/c-inst(y, p)

(4). **if** $c/y{\in}x$, $c/x :: p$, $c \models$ setof(p^*, q) **then** c/c-inst(y, q)

(5). **if** $c/y{\in}x$, $c/x :: p$, $c \models$ listof(p^*, q) **then** c/c-inst(y, q)

(6). **if** $c/y{\in}x$, $c/x :: p$, $c \models$ bagof(p^*, q) **then** c/c-inst(y, q)

(7). **if** $c/x[i] = y$, $c/x :: p$, $c \models$ listof(p^*, q) **then** c/c-inst(y, q)

(8). **if** $c/y{\in\in}x{=}l$, $c/x :: p$, $c \models$ bagof(p^*, q) **then** c/c-inst(y, q)

(9). **if** $c/x[i] = y$ **then** c/c-inst(i, uldInteger)

(10).**if** $c/y{\in\in}x{=}l$ **then** c/c-inst(l, uldInteger)

(11).**if** $\mathcal{K}_l^{v_1,\ldots,v_n}(v)^t$ **then** c/c-inst(v, t)

Each elaboration rule is applied directly to formulas in the head of the mapping except for the last rule (which is a special case). In particular, the last rule is matched against formulas in the body of the mapping and is only applied if v is used in a formula in the head of the mapping.

To illustrate, each formula in the consequent of the first elaboration rule holds for the mapping in Example 12. That is, T.hasName:N (from the rule body), T::entityType (from the annotation rule), and entityType.hasName->uldString (from the configuration) are each true. Thus, the formula in the consequent of the elaboration-rule, c-inst(N, uldString), is added to the head of the mapping.

Finally, we consider the more general case of filling in partial mapping rules (as opposed to filling in only missing *c-inst* relationships, as above). Here, the goal is to allow a user to specify an initial correspondence assertion [64]—for example, that E-R entity types should be mapped to relations in the relational model—and based on the correspondence enumerate the remaining details of the mapping. In general, it is not possible to automatically define the desired mapping: (1) at some point, component selector labels must be unified, and (2) data-model constraints must be used to determine the validity of and help fill-in mappings, which is not a decidable problem for all constraints expressed in stratified Datalog¬ (as discussed above). Instead of automatically generating the filled-in mapping, we sketch a procedure to fill in many, but potentially not all, details of the mapping.

The following is a list of approaches that can elaborate an initial mapping rule. Note that each approach below would be repeatedly applied to each formula in a single mapping rule $h \Longleftarrow b$ (until a fixed-point is reached), where both h and b consist of one more formulas. For example, given the statement "E-R entity types should be mapped to relations in the relational model," we would construct the initial mapping rule c-inst(E, relation) $\Longleftarrow$ c-inst(E, entityType), and apply the following techniques to fill the rule in.

Structural Elaboration Given a formula c/c-inst(x, p) in either h or b (of the mapping rule) where ct-inst(p, struct-ct) is in configuration c, we can add the formulas $c/x.s{:}t$ and c/c-inst(y, q) to h or b (where s and q are constants and t is a variable), respectively, whenever $y.s$->q is in c. We call this approach *structural elaboration*, because each component-selector definition must have a corresponding value in a construct instance.

Conformance Elaboration Given a formula c/c-inst(x, p) in either h or b where conf(p,

q, d, r) is in c and r is either '1' (exactly one) or '+' (one or more), we add the formulas c/d-inst(x, y) and c/c-inst(y, q) to h or b, respectively (where q is a constant and y is a variable). Similarly, given a formula c/c-inst(y, q) in either h or b where conf(p, q, d, r) is in c and d is either '1' (exactly one) or '+' (one or more), we add the formulas c/d-inst(x, y) and c/c-inst(x, p) to t or b_m, respectively (where p is a constant and x is a variable). We call this approach *conformance elaboration*, because each construct instance with a required cardinality constraint on conformance requires an instance of (or a type for) at least one other construct instance.

Constraint Elaboration Given the constraint e_1, ..., $e_i \Leftarrow b_{c1}$, ..., b_{cj} in configuration c, we can add the formulas c/e_1, ..., c/e_i to h if $\{c/b_{c1}, ..., c/b_{cj}\} \subseteq h$ with variable and constant substitution. In other words, if the body of the constraint matches (with unification) a subset of formulas in the head of the mapping rule, the constraint formulas e_1 to e_i are added (with variable substitution) to the head of the mapping rule. This technique can be similarly applied to the body of the mapping rule. We call this approach *constraint elaboration*, because we add information to the mapping based on available constraints.

The following six steps illustrate how the elaboration approaches above can be used to fill-in the initial mapping rule c-inst(E, relation) $\Longleftarrow$ c-inst(E, entityType). Once the rule is fully elaborated (the result of step 6), the user must still state the connection between identifiers, for example, that the relation name and the entity name should be the same (N1 and N2 below).

Step 1. Apply the initial mapping rule.

c-inst(E, relation) $\Longleftarrow$ c-inst(E, entityType).

Step 2. Apply structural elaboration to the result of step 1 to obtain the following rule.

c-inst(E, relation), E.hasName:N2, E.hasAtts:AL $\Longleftarrow$
    c-inst(E, entityType), E.hasName:N1, E.hasAtts:AS.

Step 3. Apply conformance elaboration to the result of step 2 to obtain the following rule.

c-inst(E, relation), E.hasName:N2, E.hasAtts:AL, d-inst(B, E), c-inst(B, table) $\Longleftarrow$
  c-inst(E, entityType), E.hasName:N1, E.hasAtts:AS.

Step 4. Apply constraint elaboration to the result of step 3 to obtain the following rule. Here, we apply the E-R constraint (er-36): A$\in$AS $\Leftarrow$ c-inst(E, entityType), E.hasAtts:AS.

c-inst(E, relation), E.hasName:N2, E.hasAtts:AL, d-inst(B, E), c-inst(B, table) $\Longleftarrow$
  c-inst(E, entityType), E.hasName:N1, E.hasAtts:AS, A1$\in$AS.

Step 5. Apply constraint elaboration to the result of step 4 to obtain the following rule. Here, we apply the relational constraint (rel-1): (attList isNonEmpty). Note that the constraint macro is expanded to an expression in the ULD constraint language prior to its application.

c-inst(E, relation), E.hasName:N2, E.hasAtts:AL, d-inst(B, E), c-inst(B, table),
A2$\in$AL, c-inst(A2, attribute) $\Longleftarrow$
  c-inst(E, entityType), E.hasName:N1, E.hasAtts:AS, A1$\in$AS.

Step 6. Apply structural elaboration to the result of step 5 to obtain the following rule, which is the fixed-point.

c-inst(E, relation), E.hasName:N2, E.hasAtts:AL, d-inst(B, E), c-inst(B,table),
A2$\in$AL, c-inst(A2, attribute), A2.hasName:N4, A2.hasDomain:D2 $\Longleftarrow$
  c-inst(E, entityType), E.hasName:N1, E.hasAtts:AS, A1$\in$AS, A1.hasName:N3,
  A1.hasDomain:D1.

4.4 Summary

In this chapter, we described the ULD transformation language for converting information between data models, schemas, and instance data. We used the language to describe standard data-model mappings, including E-R-to-relational and XML-to-relational transformations. We also used the transformation language to express a schema mapping, where the source and target configurations are represented in distinct data models. Finally, we outlined a number of approaches to ensure transformation mappings are valid, and described two simple approaches to elaborate partial transformation rules.

We have used the ULD transformation language to implement a number of mappings. We briefly describe our experience using the language here. In Chapter 6, we discuss ULD implementation issues, including transformation. We note that, in general, there are a number of opportunities for optimizing transformations, which we do not consider in this dissertation (see Section 6.1). However, based on our experience, we believe that transformation can be practically implemented using existing languages such as Prolog.

Our first experiments with transformation were done within the context of superimposed information models [36, 37]. Our goal was to enable interoperability for *model-based* tools. By *model-based*, we mean a tool that works for any valid information represented in a particular data model, as opposed to a tool that relies on a particular schema (in a particular data model). By defining and executing data-model mappings, our goal was to allow a user to select data of interest independently from the tool they wish to use. Thus, a user would be free to use their favorite tool over data represented in an arbitrary data model. We defined and executed a number of mappings (using Prolog) for converting information represented in the SLIMPad data model [16, 37] to information represented in the RDF and Topic-Map data models. The SLIMPad data model uses simple tree-based structures for storing data, similar to XML. We note that in these mappings, source configurations were small in size, the mapping rules did not require recursion or negation, and the transformations did not define one-to-one data-model conversions.

We also defined and executed schema mappings, where the source and target schemas were represented using different data models (XML and Topic Maps, respectively). These experiments used larger data sets, where each source configuration contained between eight- and twelve-thousand ULD facts. Each XML source contained metadata for a particular section of an on-line course, such as the section's slides (where each slide was considered an addressable, external resource), audio-video files, homework assignments, reading assignments, and so on. The XML files were used to build an HTML version of an on-line course. However, the XML files were not convenient for course developers who wished to reuse slides and other materials when developing new courses or updating existing courses. The goal of the target Topic-Map schema was to serve as a simple, conceptual model that course developers could navigate (using a Topic-Map browser) to easily find material of interest.

We originally implemented these mappings using Prolog, however, as the size of the source configurations increased, the system became slow for mapping rules with recursion and negation. (Note that only a few mapping rules used recursion.) Based on these results, we developed a prototype system, written in Java, that stored configurations in a relational database as opposed to a file of Prolog facts. We defined a relational schema to represent the ULD facts and converted mapping rules to embedded SQL programs. (Mapping rules with recursion and negation were converted to embedded SQL by hand.) Using SQL and storing configurations in a relational database improved the overall speed of executing the transformation rules.

Finally, we also used the ULD transformation language to execute schema mappings where the source and target configurations were both expressed using XML. The two schemas described similar intrusion-detection information, for example, that would be generated by an intrusion-detection system, and served as interchange formats for use in other security applications. The schemas contained various forms of structural heterogeneity (at the schema level), including different representations for data as well as structural and schematic heterogeneity. Using the ULD transformation language, we were able to declaratively describe the schema mapping, and using Prolog, execute the mapping directly to convert data between the schemas.

Chapter 5

Incremental Navigation

Like transformation, navigation is an application of the ULD. The goal of navigation is to provide generic access to structured information, much like a Web browser provides generic access to viewable information. We are particularly interested in browsing a data source where type information (if available) and structural relationships among the data items can be viewed in a step-by-step manner, where a user can select an item, select a path that leads from the item, follow the path to a new item, and so on, *incrementally* through the source. The need for incremental navigation is motivated by the following uses. First, we believe that simple browsing tools provide people with a powerful and easy way to become familiar with an information source. Additionally, having generic access to heterogeneous sources supports exploration tools used in the process of integration [25, 34]. In particular, once an information source has been identified, its contents can be examined (by a person or an agent) to determine if and how it should be combined (or integrated) with other sources.

In this section, we describe a generic set of incremental navigation operators that are implemented against the ULD, which enables various styles of browsing across a wide range of data models. In addition, the ULD provides multiple ways to specify how navigation operators should work with various data models. We consider both a low-level approach for creating detailed and complete specifications as well as a simple, high-level approach for semi-automating each operator's implementation.

5.1 Motivating Examples

Incremental navigation can help an information agent (such as a crawler) or a person to become familiar with a new data source. When an information agent discovers a new source (that uses the schema, for example, in Figure 1.3), it may wish to know: (1) what data model is used (that is, is it an RDF, XML, Topic Map, or relational source?), (2) assuming RDF is used, whether any classes are defined for the source (what is the source schema?), (3) which properties are defined for a given class (what properties does the *film* class have?), (4) which objects exist for the class (what are the instances of the *film* class?) and (5) what kinds of values exist for a given property of a particular object of the class (what *actor* objects are involved in this *film* object?). The agent moves step-by-step through the source to gather information. It starts by determining what kind of source it is, and moves down through the schema to individual data items, following their links to other items.

This example assumes the user (or agent) understands the data model of the source. For example, the agent begins navigation by asking the source which RDF classes it stores. If the data model used was XML instead of RDF, the agent (assuming it understands the XML model) could have started navigation by asking for all of the available element types. We call this approach *data-model-aware browsing*, in which the constructs of the data model can be used to guide navigation.

We primarily focus in this chapter on providing incremental navigation to users who may not understand the underlying representation scheme (although we do consider data-model-aware browsing as well). We call this *direct browsing*, because the data and schema items of the source are navigated directly. Using direct browsing, our previous example would change as follows. Upon discovering an information source, the crawler might wish to know: (1) what kind of information the source contains, which (in our example) would include “films,” “actors,” and “awards,” etc., (2) (assuming the crawler is interested in films) the things that describe films, which would include “titles” and relationships to awards and actors, (3) the available films in the source, and (4) the actors of a particular film, which is obtained by stepping across the “involved” link for the film in question.

Thus, without requiring any knowledge of the source's data model, with direct navigation, the agent can still freely browse the source to gather information.

5.2 Navigation Operators

Incremental navigation refers to the ability to step iteratively through an information source one item at a time. For example, when situated at a particular item (a *location*), we can use incremental operators to determine the set of available links from the current position and follow one of these links to a new location. This style of navigation is in contrast to approaches that navigate information using path expressions [29, 45], which describe collections of paths and variable bindings over a database. Path expressions enhance query capability, whereas our goal is to enable step-by-step navigation through an information source. The following issues motivate our choice of operators.

- *How should items be represented?* We wish to support navigation regardless of the data model used by the source. However, sources represent information items using different data-model constructs. At issue is whether to expose atomic (for example, text strings) or structured values (for example, records or lists) through navigation operations. We define the navigation operators such that each operator takes and returns atomic values, which keeps the navigation interface simple, and does not require additional knowledge of the underlying structures of data models.

- *How should navigation start?* By incremental navigation, we mean navigating from a current location to a new location. However, when a source is first encountered, it is not immediately clear from which item to start navigation. Path-based approaches typically use available collections or collections formed from queries to apply navigation expressions. Because the items for beginning navigation depend on the individual source and its data model, we choose to include an operator for obtaining the starting locations of sources (that is, the operator returns one or more *entry points*).

- *How should schema-instance relationships be handled?* Data models can have different requirements on schema-data conformance and these constraints have implications for navigation. For example, a partial-schema data model may require support for navigating instances that are not included in the schema, because the schema alone cannot be used to navigate all available data. We define a minimal set of generic operators that can be used to access the various styles of schema arrangements possible in data models.

- *What information within a source should be navigable?* For a given source, some information may be meaningful for navigation, while other information may not. For example, data-model-aware navigation might offer navigable items that are not necessarily useful in direct navigation. We allow different implementations for each navigation operator through navigation bindings, expressed using the ULD query language.

The incremental navigation interface consists of the following operators. Navigable items in an information source are separated into disjoint sets of *links* and *locations*. For a given configuration, we assume a finite set of location names $\mathcal{L}$ and a finite set of *link* names $\mathcal{N}$ where both $\mathcal{L}$ and $\mathcal{N}$ consist of simple, atomic string values.

- **Starting Points**. The operator *sloc* : $\mathcal{P}(\mathcal{L})$ returns all available entry points into an information source. Note that we require the result of *sloc* to be a set of locations (as opposed to links).

- **Links**. The operator *links* : $\mathcal{L} \rightarrow \mathcal{P}\ (\mathcal{N})$ returns all out-bound links available from a particular location. We note that for some locations, there may not be any links available, that is, the *links* operator may return the empty set.

- **Following Links**. The operator *follow* : $\mathcal{L} \times \mathcal{N} \rightarrow \mathcal{P}(\mathcal{L})$ returns the set of locations that are at the end of a given link. We use the *follow* operator to prepare to move to a new location from our current location. Note that following a link may lead to a set of locations. Given the set of locations returned by the *follow* operator, the user or agent directing the navigation can choose one as the new location.

- **Types**. The operator *types* : $\mathcal{L} \rightarrow \mathcal{P}(\mathcal{L})$ returns the (possibly empty) set of types for a given location. We use the *types* operator to obtain locations that represent the schema for a data item. Note that a particular location may have zero, one, or many associated types.

- **Extents**. The operator *extent* : $\mathcal{L} \rightarrow \mathcal{P}(\mathcal{L})$ returns the (possibly empty) set of instances for a given location. The *extent* operator computes the inverse of the *types* operator.

We call a group of navigation-operator implementations (that is, one implementation for each operator) a *navigation binding* (or *binding* when the context is clear). To issue the *sloc* operation with binding b on configuration c, we use the notation $b[c].sloc$. To illustrate, consider the following binding B_{rdf} for navigating RDF sources (that is, this binding returns data-model information for data-model-aware browsing). The following shows a sequence of calls to B_{rdf}'s operator implementations and the results for a configuration named c.

$B_{rdf}[c]$.sloc = {'class', 'property', 'subClass', 'resource', 'triple', ...}
$B_{rdf}[c]$.extent('class') = {'film', 'comedy', 'thriller', ...}
$B_{rdf}[c]$.links('thriller') = {'title'}
$B_{rdf}[c]$.extent('thriller') = {'#m1', ...}
$B_{rdf}[c]$.follow('#m1', 'title') = {'The Usual Suspects'}

The goal of the operators is to allow navigation through an information source incrementally, as opposed to issuing special-purpose queries to obtain the same information. Note that, as shown by the *follow* operator in the example above, bindings present a view of the data that does not require the user to connect the various underlying data structures (for example, by joining various structures together)—in this case, the *follow* operator binding computes the connection between a property name 'title' and the use of the property within a triple (to obtain a new location). The challenge of providing incremental navigation is in specifying bindings of general use for navigation, and simplifying the effort on the part of the user for stepping through a source.

5.3 Navigation Bindings as ULD Queries

We present two ways that bindings can be implemented against the ULD. The first method treats a binding as a set of queries using the ULD query language. We call these *binding queries*. The second method provides for higher-level description of bindings (and is described in the next section).

For the query approach, we treat an operator binding as the result of a ULD query where each operator is represented as an intensional predicate in a Datalog program. We represent the incremental navigation operators using the following intensional predicates.

- sloc(r), where r represents a starting location.
- links(l, r), where r is a link from location l.
- follow(l, p, r), where r is the location that is found by following link p from some location l.
- types(l, r), where l is a location of type r.
- extent(l, r), where r is in the extent of l.

For a given operator invocation, the binding computes the least fix-point of the operator's predicate over the configuration, and returns the associated set of results. The following set of rules define a portion of the binding B_{rdf} above. We note that the definition uses an additional intensional predicate for computing the transitive closure of the RDF subclass relationship.

```
sloc(R)           ← ct-inst(R, C).
extent(class, R)  ← c-inst(R, class).
links(L, R)       ← c-inst(C, class), C.hasLabel:L, c-inst(P, property), P.hasLabel:R,
                    P.hasDomain:C′, subClassClosure(C, C′).
extent(L, R)      ← c-inst(C, class), C.hasLabel:L, d-inst(R, C).
follow(L, P, R)   ← c-inst(T, triple), T.hasPred:X, T.hasSubj:L, T.hasObj:R,
                    c-inst(R, literal), d-inst(L, C), c-inst(X, property), X.hasLabel:P,
                    X.hasDomain:C′, subClassClosure(C, C′).
```

```
subClassClosure(C, C)    ← c-inst(C, class).
subClassClosure(C1, C3) ← c-inst(S, subClass), S.hasSub:C1, S.hasSuper:C2,
                          subClassClosure(C2, C3).
```

In addition to data-model-aware navigation, we can also define operator bindings for direct navigation using the same approach. For example, the following binding queries present a complete-schema view of an RDF configuration. As shown, the only starting locations are classes and the only available links from a class are its associated properties and its associated instances. (Note that we reuse the *subClassClosure* intensional predicate defined above.)

```
sloc(R)          ← c-inst(C, class), C.hasLabel:R.
links(L, R)      ← c-inst(C, class), C.hasLabel:L, c-inst(P, property), P.hasLabel:R,
                   P.hasDomain:C′, subClassClosure(C, C′).
links(L, R)      ← c-inst(C, class), d-inst(O, C), O.hasURI:L, c-inst(T, triple),
                   T.hasPred:X, T.hasSubj:O, c-inst(X, property), X.hasLabel:R,
                   X.hasDomain:C′, subClassClosure(C, C′).
follow(L, P, R) ← c-inst(C, class), C.hasLabel:L, c-inst(T, property), T.hasLabel:P,
                   T.hasDomain:C′, T.hasRange:C″, C″.hasLabel:R,
                   subClassClosure(C, C′).
follow(L, P, R) ← c-inst(C, class), C.hasLabel:L, c-inst(T, property), T.hasLabel:P,
                   T.hasDomain:C′, T.hasRange:R, R=‘literal’,
                   subClassClosure(C, C′).
follow(L, P, R) ← c-inst(C, class), d-inst(O, C), O.hasURI:L, c-inst(T, triple),
                   T.hasPred:X, T.hasSubj:O, T.hasObj:R, c-inst(R, literal),
                   c-inst(X, property), X.hasLabel:P, X.hasDomain:C′,
                   subClassClosure(C, C′).
follow(L, P, R) ← c-inst(C1, class), d-inst(O1, C1), O1.hasURI:L, c-inst(T, triple),
                   T.hasPred:X, T.hasSubj:O1, T.hasObj:O2, O2.hasURI:R,
                   c-inst(X, property), X.hasLabel:P, X.hasDomain:C′,
                   subClassClosure(C, C′).
extent(L, R)     ← c-inst(C, class), C.hasLabel:L, d-inst(O, C), O.hasURI:R.
type(L, R)       ← c-inst(C, class), C.hasLabel=R, d-inst(O, C), O.hasURI:L.
```

$$
\begin{array}{rcl}
\textit{RDF Binding } B & = & (L, N, S, F) \\
L & = & \text{class, resource, literal} \\
N & = & \text{property, triple} \\
S & = & \text{class} \\
F & = & \text{triple : resource[triple.hasSubj]}\Rightarrow\text{literal[triple.hasObj],} \\
 & & \text{triple : resource[triple.hasSubj]}\Rightarrow\text{resource[triple.hasObj],} \\
 & & \text{property : class[property.hasDomain]}\Rightarrow\text{class[property.hasRange]} \\
\text{name(X,N)} & \leftarrow & \text{c-inst(X,class), X.hasLabel:N.} \\
\text{name(X,N)} & \leftarrow & \text{c-inst(X,resource), X.hasURI:N.} \\
\text{name(N,N)} & \leftarrow & \text{c-inst(N,literal).} \\
\text{name(X,N)} & \leftarrow & \text{c-inst(X,property), X.hasLabel:N.} \\
\text{name(X,N)} & \leftarrow & \text{c-inst(X,triple), X.hasPred=R, c-inst(R,property), R.hasLabel=N.}
\end{array}
$$

Figure 5.1: A high-level binding for direct navigation of RDF.

5.4 Higher-Level Binding Specifications

The binding-queries approach is a low-level mechanism for providing detailed and complete descriptions of bindings. Here we describe a higher-level mechanism to specify bindings, which we call the *binding constructs* approach. This approach is used for specifying direct-navigation bindings and allows a user to select certain constructs as *locations* (to be viewed) and certain other constructs as *links* (to be followed). Thus, a user can easily restrict the subset of links and locations as well as how they are traversed without having to write Datalog rules. Using this specification, the navigation operators are automatically computed by traversing the appropriate instances of locations and links in the configuration. For example, to compute the *links* operator from a given location, we start from the location's construct instance and consider connections from this instance to other instances to see if any are designated as a link. Figure 5.1 shows an example of a binding-constructs definition for RDF. In the example, RDF classes, resources, and literals are considered sources for locations and RDF properties and triples are considered sources for links.

We define the binding-constructs specification as a tuple of the form (L, N, S, F). The disjoint sets L and N consist of construct identifiers such that L is the set of constructs used as locations and N is the set of constructs used as links. The set $S \subseteq L$ gives the entry points of the binding. Finally, the set F contains *link definitions* (described below), which fill in the details of how links should be traversed.

Each construct in L and N has an associated naming definition that describes how to compute the name of an instance of the construct. The name would typically be viewed by the user during navigation. (Recall that the incremental operators work over string values, therefore, the naming definitions serve to map location and link instances to appropriate string values.) A naming definition is expressed as a Datalog rule whose head is the intensional predicate name(x,n) such that x is a given construct instance and n is the string to be used as the name of the instance. For example, in Figure 5.1, RDF classes and properties are named by their labels, resources are named by their URI values, a literal value is used directly as its name, and the name of a triple is the name of its associated predicate value.

The incremental operators in the binding-constructs approach are computed automatically by traversing connected instances. We define the following rules to compute when two instances are connected. (Note that these connected rules only perform single-step traversal, and in general, can be extended to allow an arbitrary number of steps.)

connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, struct-ct), X_1.S:X_2.
connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, bag-ct), $X_2 \in X_1$.
connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, list-ct), $X_2 \in X_1$.
connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, set-ct), $X_2 \in X_1$.
connected(X_2, X_1) ← connected(X_1, X_2).

A connected formula is true when there is a structural connection between two instances. Note that the rules above do not consider the case when two items are linked by a *d-inst* relationship. Instead, the *d-inst* relationship is directly used by the *types* and *extent* operators, whereas connections are used by the *links* and *follow* operators.

The link definitions of F serve to describe the behavior of links. A link definition in F has the form $c_k : c_1[p_1] \Rightarrow c_2[p_2]$ where:

- The behavior of the link construct $c_k \in N$ is being described by the rest of the expression. For example, in Figure 5.1, the first link definition is for triple constructs.

- For $c_1, c_2 \in L$, the construct c_k can serve to link an instance of c_1 to an instance of c_2. Thus, we can traverse from instances of c_1 to instances of c_2 via an instance of c_k. For example, in Figure 5.1, the first link definition says that we can follow a resource instance to a literal instance if they are connected by a triple.

- The expressions p_1 and p_2 further define how instances of c_k can be used to link c_1 and c_2 instances, respectively. For example, in Figure 5.1, the first link definition states that triples link resources and literals through the triple's hasSubj and hasObj selector, respectively.

Link definitions are used directly to define the *links* and *follow* operations. We introduce the *linkSource*(F, i_1, i_k) and *linkTarget*(F, i_k, i_2) clauses to filter possible connections that are considered when computing the *links* and *follow* operations. Given a set of link definitions F and a connection from i_1 to i_k such that connected(i_1, i_k) is true (where i_k is the link instance), *linkSource*(F, i_1, i_k) is true if there is at least one link definition $c_k : c_1[p_1] \Rightarrow c_2[p_2]$ in F such that i_1 is an instance of c_1 (that is, c-inst(i_1, c_1) is true), i_k is an instance of c_k (that is, c-inst(i_k, c_k) is true where c_k is a link construct), and i_1 and i_k are connected according to the expression p_1. Similarly, *linkTarget*(F, i_k, i_2) is true if there is at least one link definition $c_k : c_1[p_1] \Rightarrow c_2[p_2]$ in F such that i_k is an instance of c_k (a link construct), i_2 is an instance of c_2, and i_k and i_2 are connected according to the expression p_2.

Given the above definitions, we automatically compute each navigation operator using the Datalog queries in Figure 5.2. We assume each operator is represented as an intensional predicate (as before) and the binding specification $B = (L, N, S, F)$ is stored as a set of extensional predicates $G_l(X)$, $G_n(X)$, $G_s(X)$, and $G_f(X)$. For example, the expression $G_l(X)$ binds X to L in the binding B. We also assume that the *name* predicate is stored as an intensional formula (defined in the binding).

The first rule finds the set of entry points: It obtains a construct in the set of starting locations, finds an instance of the construct, and then computes the name of the instance. The second rule finds the locations with links. For each named instance in the configuration that is connected to another instance, we use the *linkSource* predicate to check if it is a valid connection, we check to make sure that the link instance (represented as the variable Y) is valid, and then compute the name of the instance. The third rule is similar to the

sloc(R) $\leftarrow$ G_s(S), P$\in$S, c-inst(X, P), name(X, R).
links(L, R) $\leftarrow$ name(X, L), connected(X, Y), G_f(F), linkSource(F, X, Y), c-inst(Y, P), G_n(N), P$\in$N, name(P, R).
follow(L, P, R) $\leftarrow$ name(X, L), connected(X, Y), G_f(F), linkSource(F, X, Y), c-inst(Y, Q_1), G_n(N), $Q_1\in$N, name(Q_1, P), connected(Y, Z), c-inst(Z, Q_2), G_l(L), $Q_2\in$L, linkTarget(F, Y, Z).
type(L, R) $\leftarrow$ name(X, L), c-inst(X, P), G_l(L), P$\in$L, d-inst(X, Y), c-inst(Y, Q), Q$\in$L, name(Y, R).
extent(L, R) $\leftarrow$ name(Y, L), c-inst(Y, Q), G_l(L), Q$\in$L, d-inst(X, Y), c-inst(X, P), P$\in$L, name(X, R).

Figure 5.2: The Datalog rules to compute links and locations based on a binding-constructs specification.

second, except it additionally uses the *linkTarget* to determine the new location. Finally, the last two rules use the *d-inst* relationship to find types and extents.

To demonstrate the approach, we use the binding definition of Figure 5.1 and the sample configuration of Figure 5.3. This configuration shows part of Figure 1.3 as a graph whose nodes are construct instances and edges are either connections between structures or *d-inst* links. Consider the following series of invocations, assuming the configuration is named *c*.

1. $B[c]$.*sloc* = {'film','thriller'}. According to the binding definition, the *sloc* operator returns all the labels of *class* construct instances. As shown in Figure 5.3, the only *class* construct instances are *thriller* and *film*.

2. $B[c]$.*links*('film') = {'title'}. The *links* operator is computed by considering each connected construct instance of *film* until it finds a construct instance whose associated construct is in N. As shown, the only such instance is *title*, which is an RDF property.

3. $B[c]$.*extent*('film') = {'#m1'}. The *extent* operator looks for the *d-inst* links of the given instance. As shown, the only such link for *film* is to *m1*.

4. $B[c]$.*follow*('#m1', 'title') = {'The Usual Suspects'}. The *follow* operator starts in the same way as the *links* operator by finding instances (whose constructs are in N)

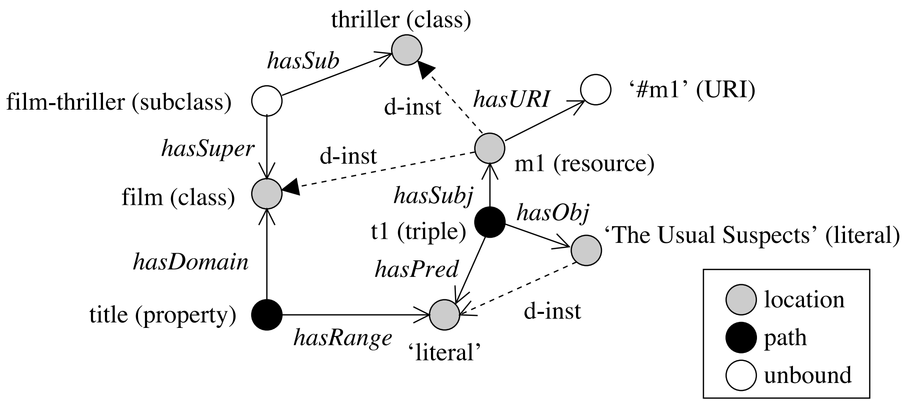

Figure 5.3: An example of labeled instances for RDF.

that are connected to the given instance. For the given item, the only such instance in Figure 5.3 is *t1*. The *follow* operator then returns the *hasObj* component of *t1*, according to the link definition for *triple*.

Note that according to the rules for computing the *follow* and *links* operators, we only consider instances that are directly connected to each other (through the *connected* predicate). Thus, invoking the operator links('thriller') will not return a result (for our example) because there are no link construct instances directly connected to the associated RDF class. However, because of the implied relationship among subclasses, the algorithm should also include the *title* link, because *thriller* is a subclass of *film*. That is, the properties of a class in RDF also include the properties of its subclasses. One way to include such information is to expand the set of connection rules to consider subclass instances. The connection rules could be augmented to consider instances that are indirectly connected through intermediate instances (for example, for certain cases such as subclass). These new connection definitions could be added directly to the binding definition of the RDF data model. Alternatively, it may be possible to extend the binding specification to include multiple steps, for example, through arbitrary path expressions in F, which could be used to define the appropriate connected rules.

Using the ULD, we have built a prototype tool for data-model-aware browsing called

JustBrowsing [18]. In JustBrowsing, we implemented a graphical user interface and the navigation operations to allow users to browse information stored in the RDF, relational, XML, and Topic-Map data models (see Section 6.1). Although not discussed here, we believe that by allowing multiple-step connections, binding-constructs specifications can be given for XML and relational data models, among others. For example, in the relational model, locations could be represented using relations, tuples, and values, and links could be represented using foreign keys. Similarly, for XML, locations could be represented using elements, element definitions, pcdata, and cdata, and links could be represented using attributes and subelement relationships. As future work, we intend to define navigation for additional data models using binding-constructs specifications.

Finally, we believe the approach described in this chapter provides a simple, generic abstraction (that consists of links and locations) that can be applied over arbitrary data models. More than that, with the binding-constructs approach, it becomes relatively straightforward to specify the links and locations for a data model, thus enabling generic and uniform access to information represented in any underlying data model (represented in the ULD). We believe that this approach can be extended beyond navigation to include querying information sources (that is, querying links and locations) and for specifying high-level mappings between data sources.

Chapter 6

Conclusion

Data models provide the basic data structures used to organize and store data within an information management system. Data models differ in their choice of structures, constraints, and schema requirements. The use of different data models results in structural heterogeneity, which can also include differences in schema (often due to the use of heterogeneous data models) and in how data values are represented. Structural heterogeneity must be resolved to combine information from multiple sources and to enable tool interoperability.

Most approaches for managing structural heterogeneity [1, 10, 24, 35, 40, 46, 57, 62, 64, 68, 69, 70, 75] focus on differences between schemas and assume a common data model is used for schema mapping and integration. Alternatively, a meta-data-model can be used to explicitly represent differences in data models. Many meta-data-model approaches [5, 6, 8, 31, 32, 78, 30, 63] provide languages for converting schema and corresponding instance data in one source to valid schema and instance data in another source (thus supporting one style of data-model mapping). These approaches, however, are characterized by their assumption that schema is required by all data models. Because of this assumption, these approaches cannot accurately describe more flexible data models, such as XML and RDF, cannot capture the various conformance constraints inherent in data models, and cannot express many of the desired mappings between sources that use agreed-upon conventions for representation or that must leverage multiple levels of information in a source.

The Uni-Level Description (ULD) extends existing meta-data-models by separating schema and instance constructs (through three distinct instance-of relationships) and by providing uniform access to data model, schema, and instance data. The ULD can describe a wide variety of data models including those that allow optional, partial, and multiple levels of schema, and can be used to describe inherent data-model constraints. Finally, the ULD enables flexible transformations among information sources due to its uniform representation of data model, schema, and data.

In the rest of this chapter, we discuss our experiences with implementing the ULD for transformation and navigation, compare the ULD with other meta-data-model approaches, highlight the contributions of the ULD, and discuss possible directions for future research.

6.1 Implementation Issues

The ULD framework evolved from our earlier work [16, 17, 19, 20, 37], which uses a simpler meta-data-model primarily consisting of record-based structures (*struct-ct* constructs), atomic structures (*atomic-ct* constructs), and union types (*union-ct* constructs). Much of our implementation work centers on this earlier meta-data-model. Here, we briefly discuss implementations for transformation and navigation and a possible extension that we leave for future work.

For both transformation and navigation, we treat a ULD configuration as a fully materialized view [71] over an underlying information source. That is, we convert schema and data of an information source (such as an XML document) into a corresponding set of ULD facts. We assume the data-model description is also included in each configuration. For each tool that uses a particular data model, a program is written called a ULD *extractor*, which accesses the information source of interest to create a new configuration loaded with the appropriate set of ULD facts. Along with a ULD extractor, a ULD *injector* is written (for transformation) that takes a ULD configuration and creates a new information source (for example, a new XML file or relational database).

The implementations for both transformation and navigation use the Prolog programming language. (We also experimented with an early version of transformation written in Java, that stored ULD configurations in a relational database.) Using Prolog, a ULD configuration is stored as a knowledge-base, which is a text file that contains a set of facts and (possibly) Horn-clause rules. Figure 6.1 shows the basic transformation architecture.

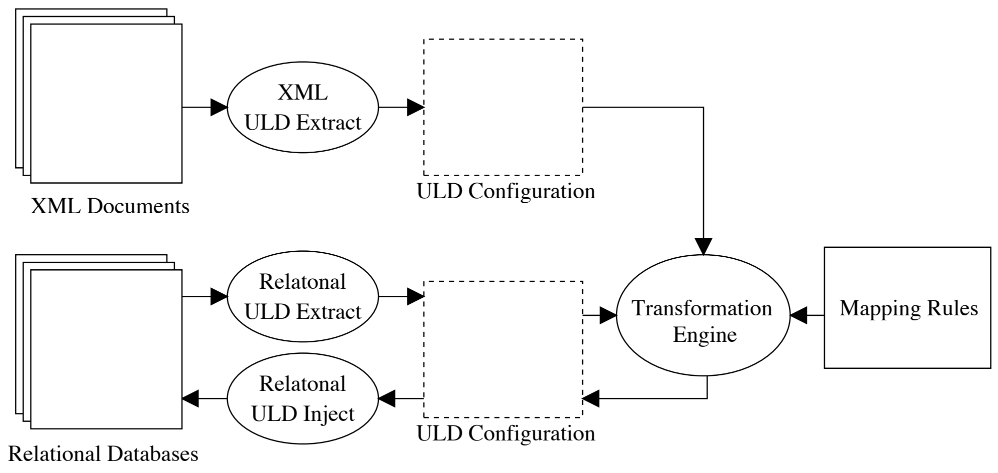

Figure 6.1: The basic architecture of the ULD transformation engine.

(The XML configuration is assumed to be the source of the transformation and the relational configuration is assumed to be the target.) In particular, we use SWI-Prolog [81], which is an open-source Prolog implementation, to execute a set of mapping rules. The transformation engine (a Prolog program) assumes there is exactly one source and one target configuration. Both the source and target configurations serve as input to the transformation engine.

Because Prolog permits only a single formula in the head of a rule, we use a special unary predicate (that takes a list of ULD formulas) to simulate the use of multiple formulas. The transformation engine first creates a new configuration to hold the result of the transformation and updates this new configuration with facts from the given target configuration. The transformation engine then uses built-in Prolog predicates to find all answers for the head formula (used to wrap multiple target formulas). The target formulas generated from the transformation are then added to the result configuration. We note that mapping rules must be written to mimic Datalog rules, for example, to ensure that all results are obtained (using cut and fail operators), and to ensure proper stratification (by manually ordering mapping rules).

Although writing mapping rules is more difficult in Prolog (because it does not explicitly compute a fixed-point), it offers some benefits. For example, in Prolog it is straightforward to write user-defined functions and most Prolog systems provide a number of built-in libraries with functions that can be used directly in mapping rules (for example, for string manipulation). Also, the SWI-Prolog system has a built-in parser for accessing XML and XML-DTD documents, RDF and RDF-Schema documents, and it supports access to relational databases through ODBC. Thus, we can write ULD extractor and injector operations (for relational, XML, and RDF sources) directly in Prolog.

Similar to transformation, applications for data-model-aware and an early version of direct-browsing are written in Prolog [18]. Figure 6.2 shows the basic architecture for the navigation implementation. We note that navigation does not require updating an information source, that is, the architecture of Figure 6.2 only requires tools for extracting underlying information into a ULD configuration and not for injecting information back to the underlying source. For browsing, we use Amzi logic server [4] (another Prolog implementation) because it can be accessed from within Visual Basic—the implementation language of the graphical interface. Thus, similar to transformation, operator binding rules are executed using Prolog.

We note that the motivation for incremental navigation, including an initial implementation, came from our work on the Superimposed-Schematics browser [21], which uses similar operators as those defined in Chapter 5 for navigation.

A potential disadvantage of using fully materialized ULD configurations is that all information in a source is initially converted to the ULD, regardless of whether it is needed for transformation or browsing. Figure 6.3 shows a modified transformation architecture in which ULD configurations are only partially materialized with data-model information. (Note that this architecture has not been implemented.) As shown, a ULD configuration in this new architecture becomes an interface that interacts directly with the underlying source. When certain facts are needed in a transformation, the transformation engine calls functions defined in the ULD interface, which then issues queries to the underlying source, converts the result to corresponding ULD facts, and returns the associated facts to the transformation engine.

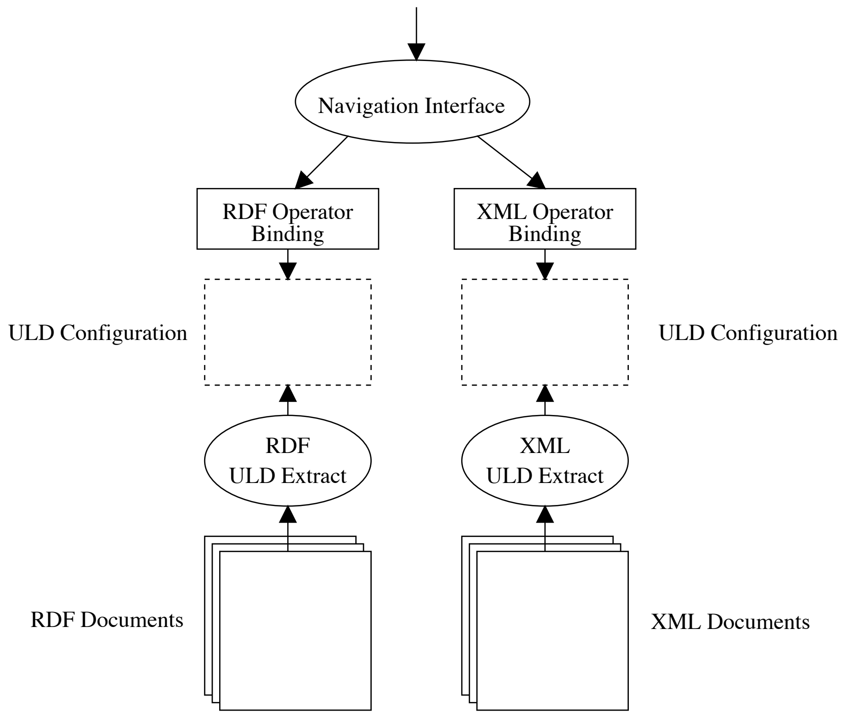

Figure 6.2: The basic architecture of the ULD navigation prototype.

The ULD interface (as shown in Figure 6.3) would contain one or more operations for each ULD formula. For example, we might have two operations for the ct-inst(x,y) formula, one to retrieve constructs x given a construct-type y, and one to retrieve a construct type y given a construct x. The ULD interface would be implemented for each tool that uses a specific data model. (Note that we can use SWI-Prolog directly to implement an interface for relational, XML, and RDF sources.) The ULD interface must also consider update operations to the underlying source for transformation.

There are a number of open issues in terms of the benefits and feasibility of using a ULD interface and partially materialized configurations. For example, one issue concerns the use of identifiers within the ULD. In the relational data model, a table construct is defined as a bag of tuple constructs. Instances of table and tuple constructs are assigned unique identifiers, which are constructed when a ULD configuration is materialized. In practice, a relational DBMS may or may not provide access to tuple identifiers. Therefore, the

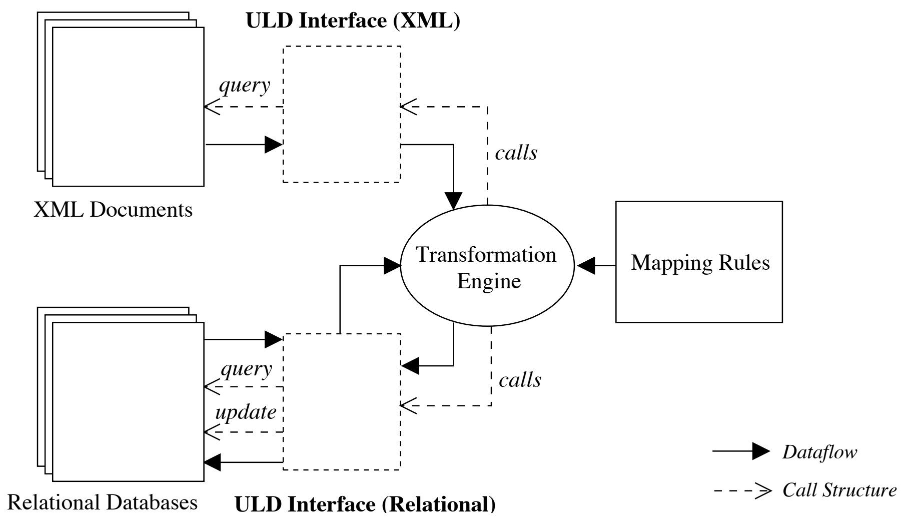

Figure 6.3: An architecture for ULD transformation using non-materialized views.

ULD interface for the relational data model must assign tuple identifiers and potentially keep track of which tuple identifiers belong to underlying tuples. Finally, we note that the use of partially materialized configurations is for efficiency. There are a number of other possibilities for providing greater efficiency, for example, using existing Datalog optimization techniques or developing techniques specifically for the ULD transformation language. In general, there are many opportunities for optimizing data-model and schema transformations.

6.2 Related Work

Here we summarize specific meta-data-model approaches related to the ULD. In general, the ULD is the only meta-data-model approach we know of that explicitly models conformance. We believe this is due, in part, to the focus of existing approaches on solving database interoperability problems, thus assuming schema is required. In addition, in most approaches, transformation is defined at the data model or schema level (but not both). The ULD enables mappings at various levels of abstraction, for example, data-model-to-data-model, data-model-to-schema, and various combinations of these. We also permit mappings that do not require complete, or one-to-one, transformations.

Atzeni and Torlone [5, 6, 78] define a meta-data-model (called MDM) to support mappings among data models to convert schemas in one data model into valid schemas in another data model. In their early work [5], they focus on eliminating the need to manually map among data models by using a *super data model.* For each data model A, two mappings are defined: (1) the constructs of A are mapped to the constructs of the super data model, and (2) the constructs of the super data model are mapped to the constructs of A. Thus, to convert a schema from data model A to B, these mappings are used to first convert the schema to the super data model, and then from the super data model to B. This approach avoids the need to map every data model to every other data model, that is, schemas from A can be converted to valid schemas of B without a direct data-model mapping from A to B.

MDM uses a set of basic structural primitives to define data-model constructs. The structural primitives are similar to E-R entity and relationship types and constructs are defined as compositions of these structures. For example, to describe an object-oriented data model, we would use the basic primitives to define the structure of classes and attributes. A schema in MDM is an instantiation of these data-model constructs. It is important to note that MDM does not consider source data. Thus, a schema type is a ground instance in MDM (that is, it cannot be further instantiated). Atzeni and Torlone also define approaches to automatically map data models based on an early meta-data-model, however, their approach assumes constructs and basic structures are one-to-one in a given data model (although structures can still be nested).

YAT [30, 32] differs from MDM in that it uses a meta-data-model to support data conversion (as opposed to schema conversion) among information sources. YAT is closely tied to XML, and is based on a hierarchical data model in which the data of an information source is represented as a tree, where nodes contain either labels or content and edges represent parent-child connections (attributes are not considered). YAT extends XML by offering a meta-data-model for defining *tree patterns*, which are used to create abstract DTDs (that is, DTDs with variables). These abstract DTDs describe the conventions (that is, tag names and tree structures) used to store information from other data models in YAT. A tree pattern allows variables as nodes and cardinalities on edges. Instance data in YAT is called a *ground pattern* because it contains constants only.

YAT uses a declarative, Horn-clause language called YATL to define schema mappings for data conversion. In YATL, rules define transformations between ground trees. However, it is not possible in YATL to define transformations at higher levels of abstraction, for example, to define mappings among data models.

When compared to MDM, YAT has a limited capability for defining the structure of data model constructs or schema. YAT is focused on defining conventions for encoding heterogeneous data as hierarchies, and does not provide an explicit description of the structures commonly used in data models.

Gangopadhyay and Barsalou [8] define a meta data model called M(DM) to support schema mapping and data conversion. M(DM) introduces a set of *metatypes*, which are similar to Atzeni and Torlone's primitive structures in that metatypes denote entities (called *things*) and relationships (called *links*). A data model in M(DM) is represented as a collection of metatypes, some of which may be introduced by the data model as specializations of default M(DM) metatypes. In M(DM), a data model construct is one-to-one with a metatype as opposed to a pattern built of basic structures as in MDM or YAT. A schema instantiates the associated data model metatypes, and a database contains instances of the corresponding schema types. Thus, metatypes are considered second-order types.

The primary goal of M(DM) is to enable data intergration in federated database systems. M(DM) provides a language called M(DM)-L for defining Horn-clause views over source schemas. M(DM) can use metatypes to uniformly access source schemas in different data models for defining global views. Thus, transformation of a source to a canonical data model is not required. However, in M(DM) the structure of a metatype is not explicit: There is no mechanism in M(DM) to range over or access the metatype definitions or structures. Therefore, M(DM) cannot support data-model mappings. We note that M(DM)-L has a similar goal as YATL, but is fundamentally different because it can use structure to access schema directly, whereas YAT-L defines views as tree transformations.

The Meta-Object Facility (MOF) standard [61] is a meta-data-model architecture proposed by the Object Management Group. The MOF is similar to MDM, however, it contains a complex and large number of primitive structures (inspired by the UML metamodel) for defining data models, called *metamodels* in MOF. The main focus of the MOF is to enable information models, such as UML class diagrams, to be shared between applications. One application of the MOF architecture is to store and interchange UML class diagrams using the XML Metadata Interchange (XMI). Given a UML model, XMI prescribes a method to encode the model as an XML DTD. In addition, the MOF has an implementation in CORBA using meta objects, for example, for describing IDL interfaces. One of the primary goals of MOF is to permit uniform access to metamodels (such as UML and IDL) and models (for example, UML class diagrams).

Finally, no other meta-data-model approach we are aware of defines a constraint language for expressing inherent data-model constraints. In the ULD, constraints can define structural restrictions (such as primary and foreign key constraints) as well as requirements on conformance. Because the ULD constraint language can be represented as a stratified Datalog$^\neg$ program, data-model constraints can also be executed to check that a configuration is correct (with respect to the given constraints). We believe, as discussed in Section 4.3, that constraints may also be used to check whether executing transformation rules will result in consistent configurations. In general, we believe that the ULD's treatment of conformance and constraints can be incorporated in and benefit current meta-data-model approaches, such as MOF.

6.3 Contributions

A key contribution of the ULD framework is its use of three, distinct *instance-of* relationships plus a conformance relationship. To the best of our knowledge, no other data-modeling or meta-data-modeling approach distinguishes type-instance relationships in this way. That is, the traditional *instance-of* or *type* relationship is not overloaded in the ULD compared with other meta-data-model approaches and data models.

The conformance relationship allows the configurer of a data model to specify the connection between a data construct definition with the corresponding schema construct definition. And by providing constraints on the conformance relation, the ULD permits accurate description of a wide range of existing data models, including traditional database data models (with strict schema requirements) and more flexible data models such as XML and RDF.

These features of the ULD framework form the basis of the contributions of this dissertation, which we highlight below.

- *A uniform representation.* The use of distinct *instance-of* relationships allows construct types, constructs, and construct instances to be treated uniformly. Thus, data model, schema, and instance data of an information source can be directly accessed, for example, within a single query, constraint, or mapping.

- *A first-order logic interpretation.* The ULD has a direct interpretation in first-order logic, and in particular, a configuration can be represented as a set of Datalog facts and accessed using standard Datalog query languages, including stratified Datalog$\neg$.

- *An expressive constraint language.* The ULD constraint language provides a convenient syntax for expressing constraints. The constraint language is a subset of full first-order logic, but is suitable for expressing most constraints for the data models we consider. In addition, the constraint language has a direct implementation using standard stratified Datalog$\neg$.

- *A language for expressing common data-model constraints.* Constraint macros offer a concise syntax for expressing many of the common constraints inherent in data models. Constraint macros have a direct conversion to equivalent expressions in the ULD constraint language.

- *A classification of conformance properties.* The ULD clarifies the schema requirements offered by existing data models by allowing these requirements to be explicitly modeled. These various schema-instance arrangements are used to define a classification scheme for the structural and process-oriented conformance properties of data models.

- *A flexible transformation language.* The ULD transformation language is based on stratified Datalog¬ and allows declarative, executable specifications of a wide variety of transformations. The transformation language can express mapping rules to convert information from one source to another, in the presence of different schemas and data models, thus providing a mechanism to resolve structural heterogeneity.

- *A generic language for incremental navigation.* The ULD offers the potential of applying generic operations—operations that are independent of a particular data model. We introduce generic operations for incremental navigation, which provide a high-level abstraction of links and locations for navigating and browsing an information source. With the ULD, we are able to implement the navigation operations using either ULD queries or high-level binding specifications so that the operations can be easily applied generically to information sources regardless of the data model used for representation.

Finally, in this dissertation, we used the ULD to: (1) express the constructs and constraints of a wide variety of data models, with varying structures and conformance properties, (2) define a number of common data-model transformations as well as a schema transformation that resolves structural heterogeneity, (3) sketch approaches to validate transformation rules and fill in partial rules, and (4) specify data-model aware and direct navigation bindings for the RDF data model.

6.4 Future Work

We describe areas of future work beyond the optimization issues discussed earlier. The following open issues are focused on transformation, and specifically on making it easier to define mappings to resolve structural heterogeneity and to further understand and exploit the details of data models and their constraints.

One reason writing transformations is difficult is that data models and schemas have various implicit constraints. Keeping track of these constraints and ensuring that a transformation does not violate them is generally difficult. Although the ULD provides an expressive constraint language (so that constraints can be explicitly captured), not all of the constraints that can be expressed can be exploited. In particular, checking that a transformation is satisfiable in the presence of constraints (that are expressed using stratified Datalog¬) is in general an undecidable problem. The constraint macros described in Chapter 3 offer one approach to limit the set of possible constraints (possibly to a decidable set). Of interest is to define a broad enough set of constraint macros to capture important data-model restrictions (such as conformance constraints), while maintaining the decidability of checking for well-formed transformations. A similar approach is used in description logics [7], where the goal is to define a decidable fragment of first order logic for constraining semantic data models. As in description logics, our challenge is to define classes of constraints that can usefully restrict data models while at the same time be practically exploited for transformation.

The ULD transformation language specifies the low-level details of a conversion. Such an approach is attractive because it enables extremely flexible transformations (for example, data-model mappings, schema mappings, and arbitrary combinations), however, specifying mapping rules is not trivial. We would like to explore the use of higher-level transformation languages that shield the low-level details of transformation. Chapter 4 sketched an informal approach for elaborating constraints, which assumes an initial correspondence between data-model constructs. Similar, correspondence-based approaches have been defined for schema mappings [64, 68, 69]. We would like to look further into applying these approaches for data-model mappings. The ULD might also provide benefits for existing schema-based approaches. For instance, if a mapping among schemas can be expressed in the ULD transformation language given initial schema correspondences, the data-model mapping would result "for free"—that is, with the ULD, we bypass the common data model, eliminating the additional conversions in and out of the common data model from source and target data models.

We also want to explore the use of transformation patterns for expressing mappings. A transformation pattern captures common mappings that can be combined to form complex structural conversions. Although not described here, we have experimented with transformation patterns [19] as parameterized specifications that take simple correspondences as input to generate complex transformations. In this approach, patterns are analogous to higher-order functions like map and fold in functional programming languages.

Another area of interest concerns modeling standard conventions with the ULD. Most data models have well-known modeling conventions for representing data structures that cannot be expressed directly with the data model's available constructs. For example, in the relational model, there are no explicit constructs for representing hierarchy; however, there are many standard conventions (or strategies) for storing hierarchical data using relations (some of these were used in the XML to relational transformations of Chapter 4). We believe the ULD can be used to describe such modeling conventions (perhaps as transformation patterns). By capturing modeling conventions and relating them to standard modeling concepts in conceptual modeling theory, it may be possible to automate data-model transformations. In addition, such a repository of conventions could serve as a guide for database design.

Bibliography

[1] Abiteboul, S., Cluet, S., and Milo, T. Correspondence and translation for heterogeneous data. In *Proceedings of the 6th International Conference on Database Theory (ICDT)* (1997), vol. 1186 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 351–363.

[2] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[3] Albert, J. Theoretical foundations of schema restructuring in heterogeneous multidatabase systems. In *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management* (2000), ACM Press, pp. 461–470.

[4] Amzi! inc. *Amzi! Prolog + Logic Server*. Available: http://www.amzi.com/, [Viewed: December 11, 2003].

[5] Atzeni, P., and Torlone, R. Schema translation between heterogeneous data models in a lattice framework. In *Proceedings of the 6th IFIP TC-2 Working Conference on Data Semantics (DS-6)* (1995), Chapman and Hall, pp. 345–364.

[6] Atzeni, P., and Torlone, R. Management of multiple models in an extensible database design tool. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)* (1996), vol. 1057 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 79–95.

[7] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., Eds. *The Description Logic Handbook*. Cambridge University Press, 2003.

[8] Barsalou, T., and Gangopadhyay, D. M(DM): An open framework for interoperation of multimodel multidatabase systems. In *Proceedings of the 8th International Conference on Data Engineering (ICDE)* (1992), IEEE Computer Society, pp. 218–227.

[9] Barta, R. AsTMa* language family. Tech. rep., Bond University, July 2003. Available: http://astma.it.bond.edu.au/astma-family.dbk, [Viewed: December 11, 2003].

[10] Batini, C., Lenzerini, M., and Navathe, S. B. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys 18*, 4 (1986), 323–364.

[11] Berners-Lee, T., Hendler, J., and Lassila, O. The semantic web. *Scientific American 284*, 5 (2001), 34–43.

[12] Bernstein, P. A., Halevy, A. Y., and Pottinger, R. A vision of management of complex models. *SIGMOD Record 29*, 4 (Dec. 2000), 55–63.

[13] Biezunski, M., Bryan, M., and Newcomb, S. R., Eds. *Topic Maps*. ISO/IEC 13250: Information Technology – Document Description and Markup Languages, December 1999. Available: http://www.y12.doe.gov/sgml/sc34/document/0129.pdf, [Viewed: December 11, 2003].

[14] Bohannon, P., Freire, J., Roy, P., and Siméon, J. From XML Schema to relations: A cost-based approach to XML storage. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)* (2002), IEEE Computer Society, pp. 64–73.

[15] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language Users Guide*. Addison-Wesley Publishing, 1998.

[16] Bowers, S., and Delcambre, L. Representing and transforming model-based information. In *Electronic Proceedings of the ECDL International Workshop on the Semantic Web* (2000), pp. 1–14. Available: http://www.ics.forth.gr/isl/SemWeb/proceedings/session1-1/paper.pdf, [Viewed: December 11, 2003].

[17] Bowers, S., and Delcambre, L. A generic representation for model-based information. *Electronic Transactions on Artificial Intelligence (ETAI) [Online] 5*, 4 (2001), 1–34. Available: http://www.ida.liu.se/ext/epa/ej/etai/2001/D/index.html, [Viewed: December 11, 2003], printed version published by *Royal Swedish Academy of Sciences*.

[18] Bowers, S., and Delcambre, L. JustBrowsing: A generic API for exploring information. In *Demo Session at the 21st International Conference on Conceptual Modeling (ER)* (2002).

[19] Bowers, S., and Delcambre, L. On modeling conformance for flexible transformation over data models. In *Knowledge Transformation for the Semantic Web* (2003), vol. 95 of *Frontiers in Artificial Intelligence and Applications*, ISO Press, pp. 34–48.

[20] Bowers, S., and Delcambre, L. The uni-level description: A uniform framework for representing information in multiple data models. In *Proceedings of the 22nd International Conference on Conceptual Model (ER)* (2003), vol. 2813 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 45–58.

[21] Bowers, S., Delcambre, L., and Maier, D. Superimposed schematics: Introducing E-R structure for *in-situ* information selections. In *Proceedings of the 21st International Conference on Conceptual Model (ER)* (2002), vol. 2503 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 90–104.

[22] Bray, T., Paoli, J., Sperberg-McQueen, C., and Maler, E., Eds. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation. World Wide Web Consortium (W3C), October 2000. Available: http://www.w3.org/TR/2000/REC-xml-20001006, [Viewed: December 11, 2003].

[23] Brickley, D., and Guha, R., Eds. *RDF Vocabulary Description Language 1.0: RDF Schema.* W3C Working Draft. World Wide Web Consortium (W3C), February 2003. Available: http://www.w3.org/TR/2003/WD-rdf-schema-20030123, [Viewed: December 11, 2003].

[24] Cali, A., Calvanese, D., Giacomo, G. D., and Lenzerini, M. On the expressive power of data integration systems. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER)* (2002), vol. 2503 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 338–350.

[25] Carey, M. J., Haas, L. M., Maganty, V., and Williams, J. H. PESTO: An integrated query/browser for object databases. In *Proceedings of 22nd International Conference on Very Large Data Bases (VLDB)* (1996), Morgan Kaufmann, pp. 203–214.

[26] Cattell, R., Ed. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

[27] Chen, P. P. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS) 1*, 1 (1976), 9–36.

[28] Chen, P. P. A preliminary framework for Entity-Relationship models. In *Proceedings of the National Computer Conference* (1977), AFIPS Press, pp. 77–84.

[29] Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. From structured documents to novel query facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (1994), ACM Press, pp. 313–324.

[30] Christophides, V., Cluet, S., and Siméon, J. On wrapping query languages and efficient xml integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2000), ACM Press, pp. 141–152.

[31] Claypool, K. T., and Rudensteiner, E. A. Sangam: A framework for modeling heterogeneous database transformations. In *Proceedings of the 5th International Conference on Enterprise Information Systems* (2003), vol. 1, pp. 219–224.

[32] Cluet, S., Delobel, C., Siméon, J., and Smaga, K. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (1998), ACM Press, pp. 177–188.

[33] Codd, E. A relational model of data for large shared data banks. *Communications of the ACM (CACM) 13*, 6 (1970), 377–387.

[34] Cohen, W. W. Some practical observations on integration of web information. In *Informal Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)* (1999), pp. 55–60.

[35] Davidson, S. B., and Kosky, A. WOL: A language for database transformations and constraints. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)* (1997), IEEE Computer Society, pp. 55–65.

[36] Delcambre, L., and Maier, D. Models for superimposed information. In *Advances in Conceptual Modeling (ER)* (1999), vol. 1727 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 264–280.

[37] Delcambre, L., Maier, D., Bowers, S., Weaver, M., Deng, L., Gorman, P., Ash, J., Lavelle, M., and Lyman, J. Bundles in captivity: An application of superimposed information. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)* (2001), IEEE Computer Society, pp. 111–120.

[38] Elmasri, R., and Navathe, S. B. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, 1994.

[39] Genesereth, M. R., and Nilsson, N. J. *Logic Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1987.

[40] Hammer, J., and McLeod, D. On the resolution of representational diversity in multidatabase systems. In *Management of Heterogeneous and Autonomous Database Systems*. Morgan Kaufmann, 1998, pp. 91–118.

[41] Hayes, P., and McBride, B., Eds. *RDF Semantics.* W3C Working Draft. World Wide Web Consortium (W3C), January 2003. Available: http://www.w3.org/TR/2003/WD-rdf-mt-20030123, [Viewed: December 11, 2003].

[42] Hendersen-Sellers, B. Advanced OO modelling: Metamodels and notations for the new millennium. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER)* (2002), vol. 2503 of *Lecture Notes in Computer Science*, Springer-Verlag, p. 11.

[43] Jarke, M., Eherer, S., Gallersdórfer, R., Jeusfeld, M. A., and Staudt, M. ConceptBase – A deductive object base manager. Tech. rep., Aachener Informatik-Berichte 93-14, 1995.

[44] Kiczales, G., Rivieres, J. D., and Bobrow, D. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[45] Kifer, M., Kim, W., and Sagiv, Y. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1992), ACM Press, pp. 393–402.

[46] Krishnamurthy, R., Litwin, W., and Kent, W. Language features for interoperability of databases with schematic discrepancies. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (1991), ACM Press, pp. 40–49.

[47] Lassila, O., and Swick, R. R., Eds. *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation. World Wide Web Consortium (W3C), February 1999. Available: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, [Viewed: December 11, 2003].

[48] Lausen, G., and Vossen, G. *Models and Languages of Object-Oriented Databases.* Addison-Wesley, 1998.

[49] Lécluse, C., Richard, P., and Vélez, F. O_2, an object-oriented data model. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (1988), ACM Press, pp. 424–433.

[50] Levy, A. Y., and Rousset, M.-C. Verification of knowledge bases based on containment checking. *Artificial Intelligence 101*, 1-2 (1998), 227–250.

[51] Levy, A. Y., and Sagiv, Y. Constraints and redundancy in Datalog. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1992), ACM Press, pp. 67–80.

[52] Ludäscher, B., Gupta, A., and Martone, M. E. Model-based mediation with domain maps. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)* (2001), IEEE Computer Society, pp. 81–90.

[53] Luger, G. F. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (4th Edition).* Addison-Wesley Publishing, 2002.

[54] Mark, L., and Roussopoulos, N. Integration of data, schema and meta-schema in the context of self-documenting data models. In *Proceedings of the 3rd International Conference on Entity-Relationship Approach (ER)* (1983), North-Holland, pp. 585–602.

[55] Marshall, C. C., Shipman III, F. M., and Coombs, J. H. VIKI: Spatial hypertext supporting emergent structure. In *European Conference on Hypertext Technology (ECHT)* (1994), ACM Press, pp. 13–23.

[56] McGuinness, D. L., and van Harmelen, F., Eds. *OWL Web Ontology Language Overview.* W3C Candidate Recommendation. World Wide Web Consortium (W3C), August 2003. Available: http://www.w3.org/TR/2003/CR-owl-features-20030818, [Viewed: December 11, 2003].

[57] Miller, R. J. Using schematically heterogeneous structures. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1998), ACM Press, pp. 189–200.

[58] Mycroft, A., and O'Keefe, R. A. A polymorphic type system for Prolog. *Artificial Intelligence 23*, 3 (1984), 295–307.

[59] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems 8*, 4 (1990), 325–362.

[60] Noy, N., Fergerson, R., and Musen, M. The knowledge model of Protege-2000: Combining interoperability and flexibility. In *Proceedings of the 2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW)* (2000), vol. 1937 of *Lecture Notes in Computer Science*, Springer-Verlag.

[61] OMG. *Meta Object Facility (MOF) Specification*, September 1997. OMG Document ad/97-08-14, Available: http://www.dstc.edu.au/Research/Projects/MOF/Publications.html, [Viewed: December 11, 2003].

[62] Papakonstantinou, Y., Abiteboul, S., and Garcia-Molina, H. Object fusion in mediator systems. In *Proceedings of 22nd International Conference on Very Large Data Bases (VLDB)* (1996), Morgan Kaufmann, pp. 413–424.

[63] Papazoglou, M. P., and Russell, N. A semantic meta-modeling approach to schema transformation. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)* (1995), ACM Press, pp. 113–121.

[64] Parent, C., and Spaccapietra, S. Issues and approaches of database integration. *Communications of the ACM 41*, 5 (May 1998), 166–178.

[65] Pepper, S., Ed. *Requirements for a Topic Map Constraint Language (Draft).* ISO/IEC JTC 1/SC34: Information Technology – Document Description and Markup Languages, April 2003. Available: http://www.isotopicmaps.org/tmcl/requirements.html, [Viewed: December 11, 2003].

[66] Pepper, S., and Moore, G., Eds. *XML Topic Maps (XTM) 1.0.* TopicMaps.Org Authoring Group, Specification, August 2001. Available: http://www.topicmaps.org/xtm/1.0/xtm1-20010806.html, [Viewed: December 11, 2003].

[67] Popa, L., and Tannen, V. An equational chase for path-conjunctive queries, constraints, and views. In *Proceedings of the 7th International Conference on Database Theory (ICDT)* (1999), vol. 1540 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 39–57.

[68] Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M., and Fagin, R. Translating Web data. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)* (2002), Morgan Kaufmann, pp. 598–609.

[69] Pottinger, R., and Bernstein, P. A. Merging models based on given correspondences. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)* (2003), Morgan Kaufmann, pp. 826–837.

[70] Rahm, E., and Bernstein, P. A. A survey of approaches for automatic schema matching. *The VLDB Journal 10*, 4 (2001), 334–350.

[71] Ramakrishnan, R., and Gehrke, J. *Database Management Systems, 3rd Edition.* McGraw Hill, 2003.

[72] Rath, H. H. The topic maps handbook. Tech. rep., empolis GmbH, 2003. Available: http://www.empolis.com/download/docs/whitepapers/empolistopicmapswhitepaper_eng.pdf, [Viewed: December 11, 2003].

[73] Reiter, R. Towards a logical reconstruction of relational database theory. In *On Conceptual Modeling*, M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, Eds. Springer-Verlag, 1984.

[74] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., and Naughton, J. Relational databases for querying XML documents: Limitations and opportunites. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)* (1999), Morgan Kaufmann, pp. 302–314.

[75] Sheth, A. P., and Larson, J. A. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys 22*, 3 (1990), 183–236.

[76] Shipman III, F. M., wei Hsieh, H., Maloor, P., and Moore, J. M. The visual knowledge builder: A second generation spatial hypertext. In *The 12th ACM Conference on Hypertext and Hypermedia (Hypertext 2001)* (2001), ACM Press, pp. 113–122.

[77] Tian, F., DeWitt, D. J., Chen, J., and Zhang, C. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record 31*, 1 (2002), 5–10.

[78] Torlone, R., and Atzeni, P. A unified framework for data translation over the web. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE)* (2001), IEEE Computer Society, pp. 350–358.

[79] Tsichritzis, D. C., and Lochovsky, F. H. *Data Models.* Prentice-Hall, 1982.

[80] Ullman, J. D. *Priniciples of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[81] University of Amsterdam. *SWI-Prolog.* Available: http://www.swi-prolog.org/, [Viewed: December 11, 2003].

[82] Woods, W. A. What's in a link: Foundations for semantic networks. In *Representation and Understanding: Studies in Cognitive Science* (1975), Academic Press, pp. 35–82.