Representing and Querying Changes in Semistructured Data*

Sudarshan S. Chawathe Serge Abiteboul Jennifer Widom

Computer Science Department, Stanford University, Stanford, California 94305

{chaw,abitebou,widom}@cs.stanford.edu

Abstract

Semistructured data may be irregular and incomplete and does not necessarily conform to a fixed schema. As with structured data, it is often desirable to maintain a history of changes to data, and to query over both the data and the changes. Representing and querying changes in semistructured data is more difficult than in structured data due to the irregularity and lack of schema. We present a model for representing changes in semistructured data and a language for querying over these changes. An important feature of our approach is that we represent and query changes directly as annotations on the affected data, instead of indirectly as the difference between database states. We describe the implementation of our model and query language. We also describe the design and implementation of a query subscription service *that permits users to subscribe to changes in semistructured information sources.*

1 Introduction

Semistructured data is data that has some structure, but it may be irregular and incomplete and does not necessarily conform to a fixed schema (e.g, HTML documents). Recently, there has been increased interest in data models and query languages for semistructured data [Abi97, BDHS96, CACS94, CGMH+94, QWG+96]. We also see increased interest in *change management* in relational and object data [GHJ96, DHR96], and in the related problem of *temporal databases* [SA86, Soo91]. However, we are not aware of any work that addresses the problem of representing and querying changes in semistructured data. As will be seen, this problem is more challenging than the corresponding problem for structured data due to the irregularity, incompleteness, and lack of schema that often characterize semistructured data. Nevertheless, our approach, based on graph annotations, is also applicable to structured graph-based data.

In this paper, we present a simple and general model, *DOEM* (pronounced "doom"), for representing changes in semistructured data. We also present a language, *Chorel*, for querying over data and changes represented in DOEM. We describe our implementation of DOEM and Chorel. We also introduce a facility that allows users to subscribe to changes in semistructured data, and we describe its design and implementation based on DOEM and Chorel.

1.1 Motivating Examples

The *Palo Alto Weekly*, a local newspaper, maintains a Web site providing information about restaurants in the Bay Area [PAW]. Most of the data in the restaurant guide is relatively static. But as often happens in database applications, we are particularly interested in the dynamic part of the data. For example, we are interested in finding out which restaurants were recently added, which restaurants were seen as improving, degrading, etc. These changes can be captured by a tool that we have implemented, called *htmldiff* [CRGMW96]. The *htmldiff* program takes two versions of a web page as input, and produces as output a marked-up copy of the web page that highlights the differences between the two versions based on their semistructured contents. Our *htmldiff* system allows users to browse the marked-up web page to view the changes, and to travel back and forth between the old and new versions of the document [CRGMW96]. Please see [C3] for details, including some sample output from *htmldiff*.

For reasonably small documents, browsing the marked-up HTML files produced by *htmldiff* to view the changes of interest is a feasible option. However, as documents get larger and changes become more prevalent and varied, one soon feels the need to use queries to directly find changes of interest instead of simply browsing. (For example, the restaurant guide page is currently more than 20,000 lines long, making browsing very inconvenient.) An example of a simple change query over the restaurant data is "find all new restaurant entries." Another example is "find all restaurants whose average entree price changed." Just as browsing databases is often an ineffective way to retrieve information, the same holds for browsing data representing changes. Thus, for this example, what we need is a query language that allows queries over changes to (semistructured) HTML pages.

As another motivating example, consider a typical information library system that contains book circulation information. Suppose we wish to be notified whenever any "popular" book becomes available where, say, we define a book as popular if it has been checked out two or more times in the past month. We could partially achieve this goal by setting a trigger on the circulation database that notifies us whenever a book is returned. However, there are two problems with this approach. First, many library information systems are legacy mainframe applications on which triggers are not available. Furthermore, even in cases where the library information system is implemented using a database system that supports triggers, a user often lacks the access rights required to set triggers on the database. Second, there is often no way to access historical circulation information, so that we cannot check whether the book being returned was checked out two or more times recently. In this application too, the data may be semistructured, especially if the library system merges information from multiple sources [PAGM96]. Thus, we again need a method to compute, represent, and query changes in the context of semistructured data.

*This work was supported by the Air Force Rome Laboratories under DARPA Contract F30602-95-C-0119 and by an equipment grant from IBM Corporation.

1.2 Overview

We are interested in the three components of a change management system, in the context of semistructured data: (1) detecting changes; (2) representing changes; and (3) querying changes. Detecting changes in semistructured data is a challenging problem in practice because many information sources that we are interested in do not provide facilities or authorization for explicit monitoring of changes (e.g., using triggers). Therefore, we are often forced to infer changes based on a sequence of data snapshots. We have studied this problem in [CRGMW96, CGM97], which describe algorithms for inferring changes from snapshots of semistructured data; we therefore do not discuss component (1) further in this paper. This paper addresses the problems associated with components (2) and (3).

Since our goal is to represent changes in semistructured data, we use as a starting point the *Object Exchange Model* (*OEM*) [PGMW95], designed at Stanford as part of the *Tsimmis* project [CGMH+94]. OEM is a simple graph-based data model, with objects as nodes and object-subobject relationships represented by labeled arcs. Due to its simplicity and flexibility, OEM can encode numerous kinds of data, including relational data, electronic documents in formats such as SGML and HTML, other data exchange formats (e.g., ASN.1), and programs (e.g., C++). The basic change operations in such a graph-based model are node insertion, update of node values, and addition and removal of labeled arcs. (Node deletion is implicit by unreachability [AQM+96].) Our change representation model, *DOEM* (for *Delta-OEM*), uses *annotations* on the nodes and arcs of an OEM graph to represent changes. Intuitively, the set of annotations on a node or arc represents the history of that node or arc.

For querying over changes we use a language based on the *Lorel* language for querying semistructured data [AQM+96]. In our language, called *Chorel* (for *Change Lorel*), we extend the concept of Lorel *path expressions* to allow us to refer to the annotations in a DOEM database. The result is an intuitive and convenient language for expressing change queries in semistructured data. Although the work in this paper is founded on the OEM data model and the Lorel language, the principal concepts are applicable to any graph-based data model (semistructured or otherwise), e.g., [BDHS96, Cat94].

Our implementation of DOEM and Chorel uses a method that encodes DOEM databases as OEM databases and translates Chorel queries into equivalent Lorel queries over the OEM encoding. This encoding scheme has the benefit that we do not need to build from scratch yet another database system; instead, we capitalize on an existing database system for semistructured data. Finally, as an important first application of DOEM and Chorel, we describe the design and implementation of a *query subscription service* that permits users to subscribe to changes in semistructured data.

1.3 Contributions

Our main contributions in this paper are as follows: (1) We motivate and develop the problem of representing and querying changes in semistructured data. (2) We present a simple and general change representation model for semistructured data. An important feature of our model is that it represents changes to a database directly as graph annotations, instead of indirectly as the difference between old and new database states. (3) We describe the syntax, semantics, and implementation of a query language over changes to semistructured data. Again, an important advantage of our query language is that it allows the user to access changes directly. (4) We describe how our prototype implements this change query language on top of an existing database system by encoding the change data and by translating change queries to ordinary queries. (5) As one application of our ideas, we describe the design and implementation of a service that permits users to subscribe to changes in heterogeneous database environments. A unique feature of our service is that it enables the user to specify very precisely (using our query language) the changes of interest.

1.4 Related Work

If we consider the general problem of representing and querying the history of a database in addition to its current state, there are two main approaches. The first approach, which we call the *snapshot-collection* approach, views the history of a database as a collection of database states (snapshots). According to this view, a change operation takes a database from one state to the next. The states are ordered, usually linearly, based on time. In addition to querying the present database state, such systems allow any other state of the database to be queried. This is the approach taken by temporal databases [SA86, Soo91]. The second approach, which we call the *snapshot-delta* approach, views the history of the database as a combination of a single database snapshot and a collection of *deltas*. According to this view, we obtain various states of the database by starting with a single snapshot and applying some sequence of deltas to it. An early, simple example of this approach is the idea of *delta relations*, used in active databases [ACT96, WC96] and trigger languages [ISO94], which represent a set of changes to a relation R using two relations R^+ and R^-, where $R^+ = R_{new} - R_{old}$, and $R^- = R_{old} - R_{new}$. More recently, this approach has been used by the *Heraclitus/H2O* project to represent changes in relational and object data [DHR96, GHJ92, GHJ96]. Our work differs from the Heraclitus/H20 work in two respects. First, we represent changes in semistructured data, not just relational and object data. Second, we present a method for querying over changes as first-class entities, as opposed to using changes to generate hypothetical states that are then queried as usual. We believe that the two approaches are complementary.

1.5 Outline of Paper

The remainder of the paper is organized as follows. Section 2 reviews the Object Exchange Model (OEM), and introduces OEM change operations and histories. In Section 3, we present our OEM-based change representation model for semistructured data, DOEM. Section 4 describes our change query language, Chorel. In Section 5, we present the encoding scheme that we use to implement DOEM and Chorel using translation. Section 6 describes the query subscription system we have implemented based on the material in Sections 3–5. We conclude in Section 7. Due to space constraints, we have omitted details from some sections of this paper; we refer the reader to the extended version [CAW97] for these details.

2 The Object Exchange Model

The *Object Exchange Model* (OEM) is a simple, flexible model for representing heterogeneous, semistructured data. (Recall that semistructured data is data that may be

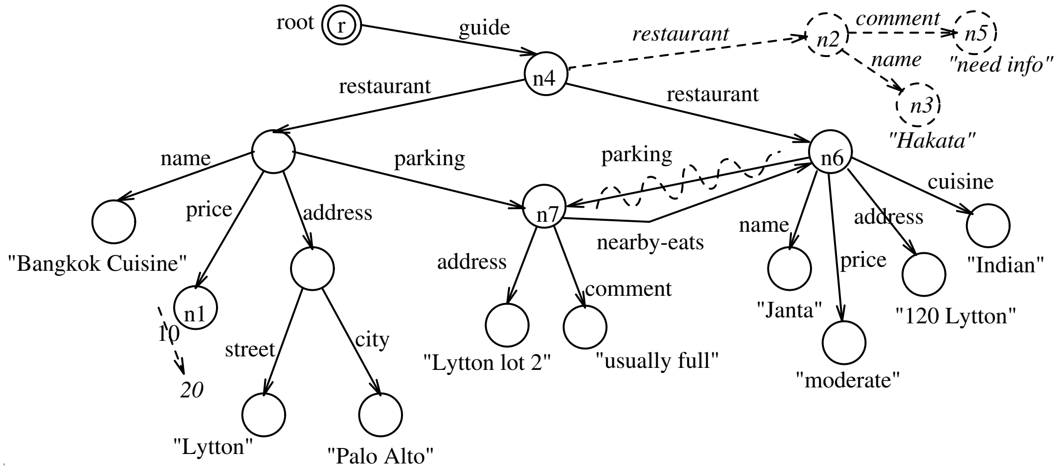

Figure 1: The OEM database in Examples 2.1 and 2.2

irregular or incomplete, and that does not necessarily conform to a fixed schema, e.g., HTML documents describing restaurants.) In this section, we begin by briefly describing OEM. Next, we define the basic change operations used to modify an OEM database. Finally, we introduce the concept of an OEM *history* that describes a collection of basic change operations. Histories form the basis of our change representation model described in Section 3.

Intuitively, one can think of an OEM database as a graph in which nodes correspond to objects and arcs correspond to relationships. Each arc has a label that describes the nature of the relationship. (Note that the graph can have cycles, and that an object may be a subobject of multiple objects via different relationships. Example 2.1 below illustrates these points.) Nodes without outgoing arcs are called *atomic objects*; the rest of the nodes are called *complex objects*. Atomic objects have a *value* of type integer, real, string, etc. An arc (p, l, c) in the graph signifies that the object with identifier c is an l-labeled subobject (child) of the complex object with identifier p. Each OEM database has a distinguished node called the *root* of the database. The root is the implicit starting point of path expressions in the Lorel query language (described in Section 4.1). Formally, we define an OEM database as follows:

Definition 2.1 An *OEM database* is a 4-tuple $O = (N, A, v, r)$, where N is a set of object identifiers; A is a set of labeled, directed arcs (p, l, c) where $p, c \in N$ and l is a string; v is a function that maps each node $n \in N$ to a value that is an integer, string, etc., or the reserved value $\mathcal{C}$ (for complex); and r is a distinguished node in N called the *root* of the database. A node is a *complex object* if its value is $\mathcal{C}$ and otherwise it is an *atomic object*. Only complex objects have outgoing arcs. We also require that every node be reachable from the root using a directed path. □

Example 2.1 We will use as our running example an OEM database describing the restaurant guide section of the *Palo Alto Weekly*, introduced in Section 1. Figure 1 shows a small portion of the data. (For this example, ignore items depicted using dashed lines.) Note that although the restaurant entries are quite similar to each other in structure, there are important differences that require the use of a semistructured data model such as OEM. In particular, we see that the price rating for a restaurant may be either an integer (10) or a string ("moderate"). The address may be either a simple string ("120 Lytton") or a complex object with subobjects listing the street, city, etc. Note also that although the data has a natural hierarchical structure, nodes may have multiple incoming arcs (e.g., node n_7), and there are cycles (e.g., the cycle formed by the arcs "parking" and "nearby-eats"). In the sequel, we refer to this database as *Guide*. □

2.1 Changes in OEM

We now describe how an OEM database is modified. Let $O = (N, A, v, r)$ be an OEM database. The four *basic change operations* are the following:

Create Node: The operation $creNode(n, v)$ creates a new object. The identifier n must be new, i.e., n must not occur in O. The initial value v must be an atomic value (integer, real, string, etc.) or the special symbol $\mathcal{C}$ (for complex).

Update Node: The operation $updNode(n, v)$ changes the value of object n, where v is an atomic value or the special symbol $\mathcal{C}$. Object n must be either an atomic object or a complex object without subobjects. (The model requires us to remove all subobjects of a complex object n before transforming it into an atomic object.) The value v becomes the new value of n.

Add Arc: The operation $addArc(p, l, c)$ adds an arc labeled l from object p (the parent) to object c (the child). Objects p and c must exist in O, p must be complex, and the arc (p, l, c) must not already exist in O.

Remove Arc: The operation $remArc(p, l, c)$ removes an arc. Objects p and c must exist in O, and O must contain the arc (p, l, c), which is removed.

If u is a basic change operation that can be applied to O, we say u is *valid* for O, and we use $u(O)$ to denote the result of applying u to O. Note that there is no explicit object deletion operation. In OEM, persistence is by reachability from the distinguished root node [AQM+96]. Thus, to delete an object it suffices to remove all arcs leading to it. A subtlety is that sometimes we need to allow objects to be "temporarily" unreachable. In particular, when we create a new object, it remains unreachable until we create an arc that links it to the rest of the database. Thus, when we consider sequences of changes in Section 2.2, we want to permit the result of atomic changes to (temporarily) contain unreachable objects. The issue is discussed further in Section 2.2 below. Note that users will typically request "higher-level" changes based on the Lorel update language [AQM+96]; the basic change operations defined here reflect the actual changes at the database level.

Example 2.2 Let us consider some modifications to the OEM database in Example 2.1. We will use these modifications as a running example in the rest of the paper. First, on January 1st, 1997, the price rating for "Bangkok Cuisine" is changed from 10 to 20. This modification corresponds to an *updNode* operation. On the same day, a new restaurant with name "Hakata" is added (with no other data). This modification corresponds to two *creNode* operations for the restaurant node and its subobject, and two *addArc* operations to add arcs labeled "restaurant" and "name." Next, on January 5th, a subobject with value "need info" is added to the "Hakata" restaurant object via an arc labeled "comment." This corresponds to one *creNode* operation and one *addArc* operation. Finally, on January 8th the parking at "Lytton lot 2" is no longer considered suitable for the restaurant "Janta," and the corresponding arc is removed; this modification corresponds to a *remArc* operation. These changes are depicted in Figure 1 using dashed lines.

2.2 OEM Histories

We are typically interested in collections of basic change operations, which describe successive modifications to the database. We say that a *sequence* $L = u_1, u_2, \ldots, u_n$ of basic change operations is *valid* for an OEM database O if u_i is valid for O_{i-1} for all $i = 1 \ldots n$, where $O_0 = O$, and $O_i = u_i(O_{i-1})$, for $i = 1 \ldots n$. We use $L(O)$ to denote the OEM database obtained by applying the entire sequence L to O. Also, we say that a *set* $U = \{u_1, u_2, \ldots, u_n\}$ of basic change operations is *valid* for an OEM database O if (1) for some ordering L of the changes in U, L is a valid sequence of changes, (2) for any two such valid sequences L and L', $L(O) = L'(O)$, and (3) U does not contain both $addArc(p, l, c)$ and $remArc(p, l, c)$ for any p, l, and c. We use $U(O)$ to denote the OEM database obtained by applying the operations in the set U (in any valid order) to O.

We are now ready to define an OEM history. Assume we are given some time domain **time** that is discrete and totally ordered; elements of **time** are called *timestamps*. Intuitively, consider an OEM database to which, at some time t_1, a set U_1 of basic change operations is applied, then at a later time t_2, another set U_2 is applied, and so on. A history represents such a sequence of sets of modifications.

Definition 2.2 An OEM *history* is a sequence $H = (t_1, U_1), \ldots, (t_n, U_n)$, where U_i is a set of basic change operations and t_i is a timestamp, for $i = 1 \ldots n$, and $t_i < t_{i+1}$ for $i = 1 \ldots n-1$. We say H is *valid* for an OEM database O if, for all $i = 1 \ldots n$, U_i is valid for O_{i-1}, where $O_0 = O$, and $O_i = U_i(O_{i-1})$ for $i = 1 \ldots n$. □

We now return to the requirement that all objects in an OEM database must be reachable from the root. An OEM history can be viewed as a sequence $L_1, \ldots, L_n$ of sequences of atomic changes. Within one sequence L_i of changes, we relax the requirement that all objects are reachable from the root so that we can, e.g., create a node and then create arcs leading to it, as discussed earlier. However, immediately after each sequence L_i has been applied, nodes that are unreachable are considered as deleted, and the remainder of the history should not operate on these objects. To simplify presentation, we also assume that object identifiers of deleted nodes are not reused.

Example 2.3 The history for the modifications described in Example 2.2 consists of three sets of basic change operations. It is given by $H = ((t_1, U_1), (t_2, U_2), (t_3, U_3))$, where $t_1 = \mathit{1Jan97}$, $t_2 = \mathit{5Jan97}$, $t_3 = \mathit{8Jan97}$, and:

$U_1 = \{$ *updNode*$(n_1, 20)$, *creNode*$(n_2, \mathcal{C})$,
creNode$(n_3,$ "*Hakata*"$)$, *addArc*$(n_4,$ "*restaurant*"$, n_2)$,
addArc$(n_2,$ "*name*"$, n_3)$ $\}$
$U_2 = \{$ *creNode*$(n_5,$ "*need info*"$)$
addArc$(n_2,$ "*comment*"$, n_5)$ $\}$
$U_3 = \{$ *remArc*$(n_6,$ "*parking*"$, n_7)$ $\}$.

This is a valid history for the original OEM database in Figure 1. □

3 Representation of Changes

In this section, we describe how changes to an OEM database are represented by attaching *annotations* to the OEM graph, thereby turning it into a *DOEM* (*Delta OEM*) graph. Intuitively, annotations are tags attached to the nodes and arcs of an OEM graph that encode the history of basic change operations on those nodes and arcs. There is a one-to-one correspondence between annotations and the basic change operations. Thus, nodes and arcs may have the following four types of annotations: (1) $cre(t)$: the node was created at time t. (2) $upd(t, ov)$: the node was updated at time t; ov is the old value. (3) $add(t)$: the arc was added at time t. (4) $rem(t)$: the arc was removed at time t. The set of all possible node annotations is denoted by **node-annot**, and the set of all possible arc annotations is denoted by **arc-annot**.

Using the above definitions of node and arc annotations, we now define a DOEM database. In the following definition, the function $f_N(n)$ maps a node n to a set of annotations on that node and the function $f_A(a)$ maps an arc a to a set of annotations on that arc.

Definition 3.1 A *DOEM database* is a triple $D = (O, f_N, f_A)$, where $O = (N, A, v, r)$ is an OEM database, f_N maps each node in N to a finite subset of **node-annot**, and f_A maps each arc in A to a finite subset of **arc-annot**. □

3.1 DOEM Representation of an OEM History

Given an OEM database O and a history $H = (t_1, U_1), \ldots, (t_n, U_n)$ that is valid for O, we would like to construct the *DOEM database representing O and H*, denoted by $D(O, H)$. $D(O, H)$ is constructed inductively as follows. We start with a DOEM database D_0 that consists of the OEM database O with empty sets of annotations for the nodes and the arcs of O. Suppose D_{i-1} is the DOEM database representing O and $(t_1, U_1), \ldots, (t_{i-1}, U_{i-1})$, for

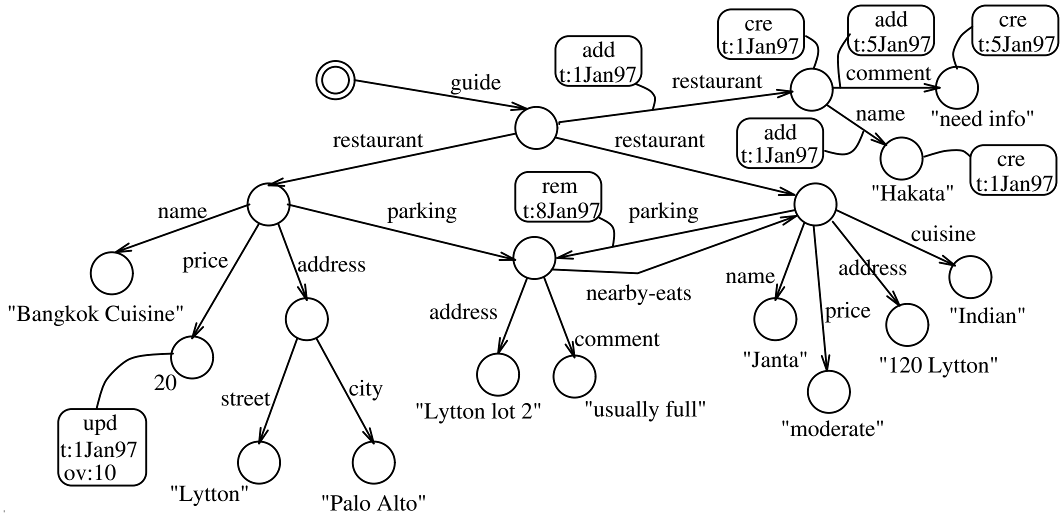

Figure 2: The DOEM database in Example 3.1.

some $1 \leq i \leq n$. The DOEM database D_i is constructed by considering the basic change operations in U_i. Since the history is valid, we can assume some ordering L_i of the operations in U_i (Definition 2.2). Starting with D_{i-1}, we process the operations in L_i in order. Whenever the value of an object is updated, in addition to performing the update we attach an *upd* annotation to the node. This annotation contains the timestamp t_i and the old value of the object. When a new object is created or an arc added, in addition to performing the modification, we attach a *cre* or *add* annotation with the timestamp t_i. When an existing arc is removed, we do not actually remove the arc from the graph; instead, we simply attach a *rem* annotation to the affected arc with the timestamp t_i. Observe that this representation directly stores the changes themselves, not the before and after images of the changes, and thus takes the *snapshot-delta* approach discussed in Section 1.4.

Example 3.1 Consider the history described in Example 2.3, which transforms the OEM database of Figure 1 as depicted there using dashed lines. The corresponding DOEM database is shown in Figure 2. We see that the DOEM database contains several annotations, depicted as boxes in the figure. For example, the annotations with timestamp "1Jan97" correspond to the first set of updates. Note that the *cre*, *add*, and *rem* annotations contain only the timestamp, while the *upd* annotation also contains the old value of the updated node (10, in our example). Also note that the removed "parking" arc from the "Janta" restaurant object to the "Lytton lot 2" parking object is not actually removed from the DOEM database; instead it bears a *rem* annotation. □

3.2 Properties of DOEM Databases

We now summarize the desirable properties of the DOEM representation of OEM database histories. (See [CAW97] for details.) Given a DOEM database D, it is easy to obtain the *original snapshot*, $O_0(D)$, the *snapshot at time t*, $O_t(D)$, and, the *current snapshot*, $O_c(D)$. It is also easy to obtain the *encoded history* $H(D)$ from a DOEM database D. We say that a DOEM database D is *feasible* if there exists some OEM database O and valid history H such that $D = D(O, H)$. Note that we do not require DOEM databases to record all changes since creation, i.e., OEM database O need not be empty. It is relatively easy to determine if a given DOEM database D is feasible. Given a feasible DOEM database D, we can show that the OEM database $O_0(D)$ and the history $H(D)$ encoded by D are unique. Thus, a DOEM database faithfully captures all the information about the history of the corresponding OEM database. Finally, as we will see in the next section, it is easy and intuitive to query the history encoded in a DOEM database.

4 Querying Over Changes

In Section 3, we have seen how the history of an OEM database is represented by the corresponding DOEM database. In this section, we describe how DOEM databases are queried. We introduce a query language called *Chorel* for this purpose. Chorel is similar to the Lorel language [AQM+96] used to query OEM databases. We begin with a brief overview of Lorel, followed by a description of the syntax and semantics of Chorel.

4.1 Lorel Overview

Lorel uses the familiar *select-from-where* syntax, and can be thought of as an extension of OQL [Cat94] in two major ways. First, Lorel encourages the use of path expressions. For instance, one can use the path expression *guide.restaurant.address.street* to specify the streets of all addresses of restaurant entries in the Guide database. Second, in contrast to OQL, Lorel has a very "forgiving" type system. When faced with the task of comparing different types, Lorel first tries to coerce them to a common type. When such coercions fail, the comparison simply returns false instead of raising an error. This behavior, while it may be unsuitable for traditional databases, is exactly what a user expects when querying semistructured data. Lorel also provides a number of syntactic conveniences such as the possibility of omitting the *from* clause. Due to space limitations, we do not describe Lorel in detail here (see [AQM+96]), but only present through a simple example those features that are needed to understand Chorel.

Example 4.1 Consider again the (modified) OEM database depicted in Figure 1. To find all restaurants that have a price rating of less than 20.5, we can use the following Lorel query:

select guide.restaurant
where guide.restaurant.price < 20.5

Note that the query expresses the price rating as a real number whereas the restaurant entries for "Bangkok Cuisine" and "Janta" in the modified OEM database shown in Figure 1 use an integer and a string, respectively. Furthermore, the third restaurant entry does not have a price subobject at all. Lorel successfully coerces the integer price 10 to real, and the comparison succeeds. For the string encoding of the price ("moderate"), Lorel tries to coerce, but fails, returning false as the result of the comparison. Finally, for the third restaurant, the missing price subobject simply causes the comparison to return false. Thus, the result of the above query is a singleton set containing the restaurant object for "Bangkok Cuisine." Note that this is an intuitively reasonable response to the original query, despite the typing difficulties and the missing data. □

4.2 Chorel

In Chorel, path expressions may contain *annotation expressions*, which allow us to refer to the node and arc annotations in a DOEM database. Informally, Lorel path expressions can be thought of as being matched to paths in the OEM database during query execution. Analogously, the annotation expressions in Chorel path expressions can be thought of as being matched to annotations on the corresponding paths in the DOEM database.

Example 4.2 Consider the DOEM database depicted in Figure 2. To find all newly added restaurant entries only, we can use the following Chorel query:

select guide.<add>restaurant

The annotation expression "*<add>*" specifies that only those objects connected to the "guide" object by a "restaurant"-labeled arc having an *add* annotation should be retrieved. For the database depicted in Figure 2, this Chorel query returns the restaurant object with name "Hakata." □

Not surprisingly, we use four kinds of annotation expressions in Chorel path expressions: *node annotation expressions* "*cre*" and "*upd*," and *arc annotation expressions* "*add*" and "*rem*." Recall that a path expression, e.g., *guide.restaurant.price*, consists of a sequence of labels. Arc annotation expressions must occur immediately before a label, whereas node annotation expressions must occur immediately after one. (Note that since node and arc annotations use different keywords, no confusion can arise.) Path expressions containing node or arc annotation expressions are called *annotated path expressions*. For instance, *guide.<add>restaurant.price<upd>* is a correct annotated path expression. It requires an *add* annotation to be present on the arc labeled "restaurant," and an *upd* annotation on the "price" node (i.e., on the node at the destination of the arc labeled "price"). For simplicity, in this paper we do not consider path expressions that have annotation expressions attached to wildcards or regular expressions, however generalizing to allow such annotation expressions is not difficult.

Annotation expressions may also introduce *time variables* to refer to the timestamps stored in matching annotations, and *data variables* to refer to the modified values in matching *upd* annotations. More precisely, the syntax of annotation expressions is as follows:

<*Annot* [*at timeV*]> if *Annot* is in { *add*, *rem*, *cre* }
<*upd* [*at timeV*] [*from oldV*] [*to newV*]> for *upd*

where *timeV*, *oldV*, and *newV* are variables. Note that a DOEM database does not explicitly store the new value of an updated object, however this information is available implicitly, and can be determined easily [CAW97].

Example 4.3 Consider the DOEM database in Figure 2. To find all restaurant entries that were added before January 4th, 1997, we can use the following Chorel query:

select guide.<add at T>restaurant
where T < 4Jan97

The Chorel preprocessor will rewrite this query to obtain the following. (We will explain this rewriting shortly.)

select R
from guide.<add at T>restaurant R
where T < 4Jan97

The introduced *from* clause will bind R to all "restaurant" objects that are connected to the "guide" object via an arc with an *add* annotation, and will provide corresponding bindings for T. More precisely, the evaluation of the *from* clause will yield the set of pairs $\langle R, T\rangle$ such that there is a *restaurant* arc from the *guide* object to R that has an *add* annotation with timestamp T. The *where* clause will filter out the $\langle R, T\rangle$ pairs for which T does not satisfy the condition. For the DOEM database in Figure 2, this query returns the restaurant object for "Hakata." □

Once time and data variables have been bound using annotations, they can be used just like other variables in Lorel or OQL. This is illustrated by the following query, which uses time and data variables in the *select* clause.

Example 4.4 Referring again to the DOEM database in Figure 2, suppose we want to find the names of all restaurants whose price ratings were updated on or after January 1st, 1997 to a value greater than 15, together with the time of the update and the new price. We can use the following query:

select N, T, NV
from guide.restaurant.price<upd at T to NV>,
guide.restaurant.name N
where T >= 1Jan97 and NV > 15

answer
name "Bangkok Cuisine"
new-value 20
update-time 1Jan97

The result of the above query is a single complex object with three components, as shown above. The label *name* is chosen by Chorel using the method described in [AQM+96]. For time and data variables whose labels are not specified by the query, Chorel chooses the default labels *create-time, add-time, remove-time, update-time, new-value*, and *old-value*. □

4.3 Chorel Semantics

We now make the semantics of Chorel queries more precise. As is done for Lorel, the semantics is described by specifying the rewriting of Chorel queries into OQL-like queries. However, we need to introduce some additional machinery to handle the annotation expressions in Chorel queries.

First, the annotation expressions in a Chorel query are transformed into a canonical form that includes all variables. For example, "*<add>*" is rewritten to "*<add at T1>*," and "*<upd from X>*" is rewritten to "*<upd at T2 from X to NV2>*," where *T1*, *T2*, and *NV2* are fresh variables. Next, as in Lorel, we eliminate path expressions by introducing variables for the objects "inside" the path expressions. For example, the path expression "*a.b.c*" in a *from* clause is converted to "*a.b X, X.c Y*," where *X* and *Y* are new *range variables*.

At this stage, we have to give a semantics to range variable definitions that may include annotation expressions (e.g., "*X.lab Y*," "*X.<add at T>lab Y*") in the context of a DOEM database. In the absence of an annotation expression, the semantics of an expression "*X.lab Y*" is that for a binding o_X of X, Y is bound to all objects o_Y such that there is an arc labeled *lab* from o_X to o_Y in the current snapshot. Note that by this semantics, a standard Lorel query (without annotations) over a DOEM database has exactly the semantics of the same query asked over the current snapshot for that DOEM database. In the presence of annotation expressions, the semantics requires the existence of the specified annotation, and also provides bindings for the variables in the annotation expression. The bindings are also specified by a special rewriting. As an example, the query in Example 4.4 is rewritten to:

select N, T, NV
from guide.restaurant R, R.price P, R.name N,
    (T, OV, NV) in updFun*(P)*
where T >= 1Jan97 and NV > 15

Our rewriting uses the following functions, which extract the information stored in annotations:

$$creFun(node) \rightarrow \{time\}$$
$$updFun(node) \rightarrow \{(time, old\text{-}value, new\text{-}value)\}$$
$$addFun(source, label) \rightarrow \{(time, target)\}$$
$$remFun(source, label) \rightarrow \{(time, target)\}$$

The function *creFun(n)* returns the set of timestamps found in *cre* annotations on node n. (Note that by our definition of change operations in Section 2.1, this set is either empty or a singleton.) The function *updFun(n)* returns a set of triples corresponding to the timestamp, the old value, and the new value in *upd* annotations on n. The function *addFun(n,l)* returns a set of (t, c) pairs such that c is an l-labeled subobject of n via an arc that has an $add(t)$ annotation. The *remFun* function is analogous to *addFun*. Once this rewriting has been performed, the *from*, *where*, and *select* clauses of the resulting query are processed in a standard manner.

Above, we have illustrated how variables introduced in the *from* clause are interpreted. Variables may be introduced in the *where* clause as well. They are treated by introducing existential quantification in the *where* clause, extending the treatment of such variables in Lorel [AQM+96]. Consider the following example:

Example 4.5 Consider again the DOEM database of Figure 2. Suppose we want the names of restaurants to which a "moderate" price subobject was added since January 1st, 1997. We can write the following Chorel query:

select N
from guide.restaurant R, R.name N
where R.<add at T>price = "moderate" and T >= 1Jan97

The variable *T* is introduced in the *where* clause. Therefore, the rewritten *where* clause is:

where exists (T, P) in addFun(R,"price") :
    (P = "moderate" and T >= 1Jan97)

□

5 Implementing DOEM and Chorel

In this section, we describe how we implement DOEM databases and Chorel queries. We encode DOEM databases as OEM databases, and we implement Chorel by translating Chorel queries to equivalent Lorel queries over the OEM encoding of the DOEM database. In addition to being more modular than a direct implementation approach that builds a Chorel database engine from scratch, this approach can also be adapted easily to other graph-based data models, e.g., those in [BDHS96, Cat94].

5.1 Encoding DOEM in OEM

Let D be a DOEM database. We encode D as an OEM database O_D defined as follows. For each object o in D, there is a corresponding object o' in O_D. An atomic object is encoded as a complex object so that we can record its history. Special labels used by the encoding start with the special character "*&*" to distinguish them from standard labels occuring in O. The encoding object o' has the following subobjects, listed by their labels.

- *&val*: If o is atomic with current value v, there is a "*&val*"-labeled arc from o' to an atomic object with value v. If o is complex, there is a "*&val*"-labeled arc from o' to itself. (This extra edge simplifies the translation of Chorel queries to equivalent Lorel queries over the encoding [CAW97].)

- *&cre*: If o has a create annotation $cre(t)$, then o' has a "*&cre*"-labeled atomic subobject with value t.

- *&upd*: For each update annotation $upd(t, ov)$ attached to o, o' has an "*&upd*"-labeled complex subobject with the following structure: a "*&time*"-labeled subobject with value t, an "*&ov*"-labeled subobject with the value before the update (ov), and a "*&nv*"-labeled subobject with the value after the update.

- *l*: If the current snapshot for D contains an arc (o, l, p), then O_D contains an arc labeled *l* from o' to the object p' that encodes p.

- *&l-history*: If D contains an arc (o, l, p), then O_D contains an arc $(o', \&l\text{-}history, o'_l)$ where o'_l is a complex object that contains the history of the l arcs from o to p. The object o'_l has the following structure: (1) *&target*: There is an arc $(o'_l, \&target, p')$, where p' is the object encoding p. (2) *&add*, *&rem*: For each annotation $add(t)$ ($rem(t)$) attached to (o, l, p), there is an "*&add*"-labeled (respectively, "*&rem*"-labeled) atomic subobject with value t.

It can be shown that all the information in a DOEM database D is fully represented in D's OEM encoding using the above scheme.

5.2 Translating Chorel to Lorel

Given the above encoding of a DOEM database as an OEM database, we now describe how a Chorel query over a (conceptual) DOEM database is translated into an equivalent Lorel query over an OEM encoding of the DOEM database. Due to space constraints, we do not present the details here, referring the reader to [CAW97] instead. The following example intuitively presents the basis of the translation scheme.

Example 5.1 Consider the Chorel query in Example 4.5. In Section 4.3, we considered the OQL-like rewriting of this query. We now complete this rewriting by using the information encoded in the &-arcs to yield the following Lorel query over the OEM encoding of the DOEM database in Figure 2:

```
select N
from guide.restaurant R, R.name N
where exists H in R.&price-history :
        exists P in H.&target :
        exists T in H.&add : T >= 1Jan97 and
                P.&val = "moderate"
```

Note that we simulate the range specification *addFun(R,"price")* using the "&"-prefixed subobjects. Further, we use *P.&val* to access the actual price value (and not the complex object packaging it with its history). □

Note that the previous query returns a set of DOEM objects that represent restaurant names. That is, it returns not only the names of the restaurants, but also the history of these names, if they changed. Returning the DOEM object enables a user interface to access both the value and the history of an object. We have implemented a DOEM database system, called CORE, based on the above ideas; please see [C3] for a description.

6 A Query Subscription Service

In Section 1, we mentioned an important application of change management: being able to notify "subscribers" of changes in (semistructured) information sources of interest to them. In this section, we describe the design and implementation of such an application, called a *Query Subscription Service* (*QSS*), using DOEM and Chorel.

An ordinary query is evaluated over the current state of the database, the results passed to the client and then discarded. An example of an ordinary query is "find all restaurants with Lytton in their address." In contrast, a *subscription query* is a query that repeatedly scans the database for new results based on some given criteria and returns the changes of interest. An example of a subscription query is "every week, notify me of all *new* restaurants with Lytton in their address." Below, we describe how subscription queries are specified and implemented in our system.

Supporting subscription queries introduces the following challenges. First, as discussed earlier, many information sources that we are interested in (e.g., library information systems, Web sites, etc.) are *autonomous* [SL90] and typical database approaches based on triggering mechanisms are not usable. Second, these information sources typically do not keep track of historical information in a format that is accessible to the outside user. Thus, a subscription service based on changes must monitor and keep track of the changes on its own, and often must do so based only on sequences of snapshots of the database states.

Briefly, our approach to constructing a query subscription service over semistructured, possibly legacy information sources is as follows: We access the information sources using *Tsimmis wrappers* or *mediators* [PGGMU95, PGMU96], which present a uniform OEM view of one or more data sources. We obtain snapshots of relevant portions of the data, and use differencing techniques based on [CRGMW96, CGM97] to infer changes based on these snapshots. Finally, we use DOEM to represent the changes, and Chorel to specify the changes of interest. We describe our approach in more detail next.

A *subscription* consists of three main components. The first component is a *frequency specification* f that specifies how often QSS should check the information source for data and changes of interest. Examples of frequency specifications are "every Friday at 5:00pm" and "every 10 minutes." The frequency specification implies a sequence of time instants $(t_1, t_2, t_3, \ldots)$, which we call *polling times*. These times are the times when we obtain a new snapshot of the data. (In the actual system [C3], we also consider two other modes: one in which the snapshots are obtained following explicit user requests, and the other in which snapshots are obtained as a result of a trigger on the source database firing, if the source provides such a triggering mechanism. To simplify the presentation, we will not consider these modes further here.

The second component of a subscription is a Lorel query Q_l, which we call the *polling query*. QSS sends the polling (Lorel) query to the wrapper or mediator at the polling times $(t_1, t_2, t_3, \ldots)$ to obtain results $(R_1, R_2, R_3, \ldots)$. An example polling query is the following. (In Lorel, "#" is a special character that matches any sequence of zero or more labels in a path, and the operator *like* performs string matching.)

```
define polling query LyttonRestaurants as
    select guide.restaurant
    where guide.restaurant.address.# like "%Lytton%"
```

Let R_0 be the empty OEM database, and let R_i be the result of the polling query on the source at time t_i for $i = 1, 2, \ldots$. Each R_i (a *Tsimmis* query result) is a tree-structured OEM database. Using differencing techniques described in [CRGMW96, CGM97], QSS obtains a history $H = (t_1, U_1), (t_2, U_2), \ldots$ corresponding to the sequence of OEM databases $(R_0, R_1, R_2, \ldots)$. That is, $U_i(R_{i-1}) = R_i$ for all $i > 0$. Then, QSS constructs a DOEM database $D(R_0, H)$ corresponding to this history H and the initial snapshot R_0, as described in Section 3. Thus, intuitively, in the first timestep the results of the polling query are all "created." Thereafter, each subsequent timestep annotates the DOEM database with the changes to the result of the polling query since the previous timestep. We identify the DOEM database corresponding to a polling query using the name of the polling query. Thus the name of the DOEM database corresponding to the above polling query is "*LyttonRestaurants*."

The third component of a subscription is a Chorel query Q_c, called the *filter query*, over the above DOEM database. In Q_c, we can use a special time variable "*t[0]*" to refer to the current polling time t_k. Similarly, we can use "*t[-1]*," "*t[-2]*," etc., to refer to the past polling times t_{k-1}, t_{k-2}, etc., respectively. (If the current polling time is t_k, we define *t[-i]* to be t_{k-i} if $i < k$, and negative infinity otherwise.) The filter query describes the data and changes of interest to the user. An example of an filter query is the following:

define filter query NewOnLytton as
select LyttonRestaurants.restaurant<cre at T>
where T > t[–1]

Given our definition of the DOEM database "*LyttonRestaurants*," this query indicates that the user should be notifed of new restaurants that have Lytton in their address since the last polling time. At each time instant t_k ($k > 0$) specified by the frequency specification, QSS evaluates Q_c over the DOEM database $D(R_0, H_k)$, where $H_k = (t_1, U_1), \ldots, (t_k, U_k)$, and returns the results to the user.

Example 6.1 Consider again the changes to the Guide data described in Example 2.2. Suppose we are interested in being notified every night of new restaurants created in the Guide database since the previous night. We issue the subscription $S = \langle f, Q_l, Q_c \rangle$, where the frequency specification f is "every night at 11:30pm," and the polling query Q_l and filter query Q_c are *Restaurants* and *NewRestaurants* (respectively) as defined below:

define polling query Restaurants as
select guide.restaurant
define filter query NewRestaurants as
select Restaurants.restaurant<cre at T>
where T > t[–1]

Suppose we create this subscription S on December 30th, 1996, at 10:00am. The polling times given by our frequency specification are $t_1 = $ *30Dec96*, $t_2 = $ *31Dec96*, $t_3 = $ *1Jan97*, and so on (all at 11:30pm). At polling time t_1, QSS sends the polling query Q_l to the Guide OEM database, to obtain the result R_1 consisting of the two original restaurant objects in Figure 1. Since R_0 is the empty OEM database by definition, both restaurant objects will have a *cre* annotation in the DOEM database built by QSS. These annotations all have a timestamp t_1, while the variable *t[-1]* in the query Q_c has value negative infinity at t_1. Therefore, evaluating the filter query Q_c on this DOEM database returns the two restaurant objects as the initial results to the user.

At polling time t_2, the Guide database is unchanged, so the result R_2 of the polling query is identical to R_1. Consequently, no changes are made to the DOEM database maintained by QSS. Note also that at time t_2, $t[-1] = t_1$, so that the create annotations on the restaurant objects in the DOEM database no longer satisfy the predicate $T > t[-1]$ in the where clause of Q_c. Therefore, the result of Q_c is empty, and the user does not receive any notification.

Before polling time t_3, the Guide database is modified by the addition of a new restaurant object, with name "Hakata," as described in Example 2.2. Therefore, at t_3, the result R_3 of the polling query contains the new restaurant object in addition to the two old restaurant objects. The new restaurant object is detected by the differencing algorithm. Accordingly, the DOEM database maintained by QSS now includes the new restaurant object, with a create annotation $cre(t_3)$ on it. Note also that at this time, $t[-1] = t_2$, so that this create annotation satisfies the predicate in the where clause of Q_c. Therefore the result of the query Q_c over the modified DOEM database contains the new restaurant object "Hakata," and the user is notified of this result. □

We have implemented our QSS prototype based on the above ideas, and interfaced it with *Tsimmis* wrappers over various information sources [C3].

7 Conclusion and Future Work

We have motivated the need for a uniform representation scheme for changes in semistructured data, and for a query language that allows direct access to changes. We have presented a simple data model, DOEM, that allows a wide variety of semistructured data to be represented together with its changes in an intuitive and compact manner. We have also presented the query language Chorel, which enables querying both the data and the changes, and described its implementation. Finally, we have described the design and implementation of a Query Subscription Service based on DOEM and Chorel.

We plan to investigate the following topics in the near future: (1) Extending Chorel to allow annotation expressions to be attached to wildcards and regular expressions in path expressions. (2) Designing indexes on annotations (based on their types and timestamps) and studying the use of such indexes to achieve a more efficient translation of Chorel queries to Lorel queries. (3) Exploring the use of *virtual annotations* to allow expressions such as *guide.restaurant.price<at t>* to refer to the value of the *price* object at time *t*, and studying their implementation. (4) Designing an event-condition-action trigger language for OEM based on ideas from DOEM and Chorel. (5) Exploring techniques to conserve space in QSS, by sharing data across subscriptions.

Acknowledgements

We are grateful to Vineet Gossain and Dan Liu for their substantial efforts in implementating Chorel and QSS, and to many members of the Stanford Database Group and the Lore project for fruitful discussions about change management.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[ACT96] ACT-NET. The active database management system manifesto: A rulebase of ADBMS features. *ACM SIGMOD Record*, 25(3):20–49, September 1996.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, November 1996.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montréal, Québec, June 1996.

[C3] Overview of the Stanford C^3 project. Available at `http://www-db.stanford.edu/c3/`.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[Cat94] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[CAW97] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data (extended version). Available at `http://www-db.stanford.edu`, 1997.

[CGM97] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis project: Integration of heterogeneous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, October 1994.

[CRGMW96] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montréal, Québec, June 1996.

[DHR96] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montréal, Québec, 1996.

[GHJ92] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Implementation of delayed updates in Heraclitus. In *Advances in Database Technology—EDBT '92, Lecture Notes in Computer Science 580*, pages 261–276. Springer-Verlag, Berlin, March 1992.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, September 1996.

[ISO94] ISO-ANSI working draft: Database language SQL3 (X3H2/94/080 and SOU/003), 1994.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Data Bases*, pages 413–424, Bombay, India, September 1996.

[PAW] The Palo Alto Weekly online edition. Embarcadero Publishing Company, Palo Alto, California. Available at `http://www.service.com/PAW/`.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 161–186, Singapore, December 1995.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A mediation system based on declarative specifications. In *Proceedings of the International Conference on Data Engineering*, pages 132–141, New Orleans, February 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[QWG+96] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A Lightweight Object REpository for semistructured data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[SA86] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[SL90] A. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[Soo91] M.D. Soo. Bibliography on temporal databases. *SIGMOD Record*, 20(1):14–24, March 1991.

[WC96] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.