# Managing Historical Semistructured Data*

**Sudarshan S. Chawathe**
*Department of Computer Science, University of Maryland, College Park, Maryland 20742.* chaw@cs.umd.edu

**Serge Abiteboul**
*INRIA—Rocquencourt, 78153 Le Chesnay Cedex, France.* Serge.Abiteboul@inria.fr

**Jennifer Widom**
*Computer Science Department, Stanford University, Stanford, California 94305.* widom@cs.stanford.edu

**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 present extensions that permit convenient snapshot-based access in our change-based data model. We also describe our design and implementation of a *query subscription service* that permits users to subscribe to changes in semistructured information sources.**


## 1. Introduction

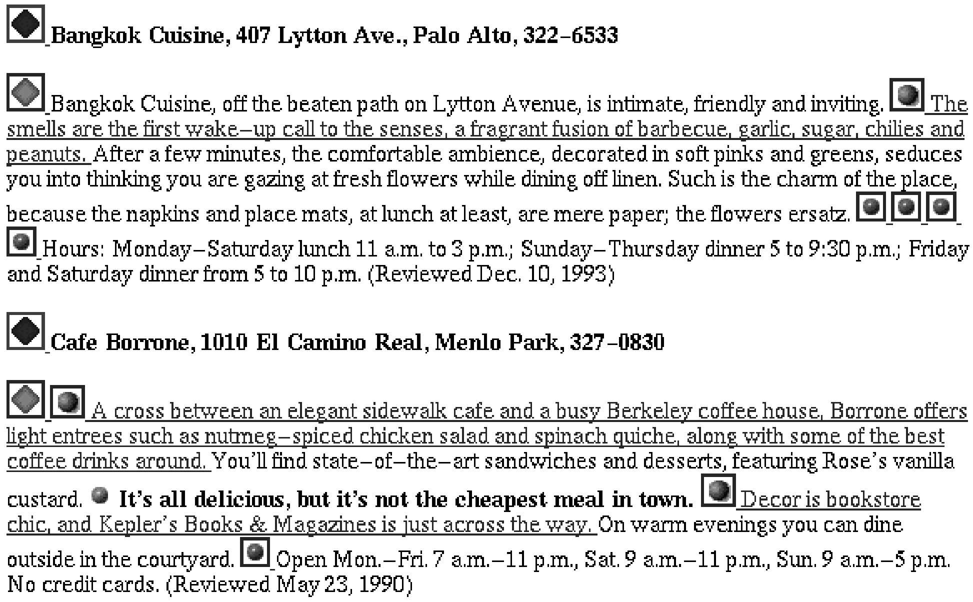
*Semistructured data* is data that has some structure, but it may be irregular and incomplete and does not necessarily conform to a fixed schema. Recently, there has been increased interest in data models and query languages for semistructured data [1, 4, 10, 8, 19]. We also see increased interest in *change management* in relational and object data [12, 11], and in the related problem of *temporal databases* [21, 22]. 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 [18]. 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* [9]. 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. A small portion of the output produced by *htmldiff* on two versions of the restaurant guide is shown in

*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. A preliminary version of this article appeared in [6].



**Bangkok Cuisine, 407 Lytton Ave., Palo Alto, 322–6533**

Bangkok Cuisine, off the beaten path on Lytton Avenue, is intimate, friendly and inviting. The smells are the first wake–up call to the senses, a fragrant fusion of barbecue, garlic, sugar, chilies and peanuts. After a few minutes, the comfortable ambience, decorated in soft pinks and greens, seduces you into thinking you are gazing at fresh flowers while dining off linen. Such is the charm of the place, because the napkins and place mats, at lunch at least, are mere paper; the flowers ersatz.

Hours: Monday–Saturday lunch 11 a.m. to 3 p.m.; Sunday–Thursday dinner 5 to 9:30 p.m.; Friday and Saturday dinner from 5 to 10 p.m. (Reviewed Dec. 10, 1993)

**Cafe Borrone, 1010 El Camino Real, Menlo Park, 327–0830**

A cross between an elegant sidewalk cafe and a busy Berkeley coffee house, Borrone offers light entrees such as nutmeg–spiced chicken salad and spinach quiche, along with some of the best coffee drinks around. You'll find state–of–the–art sandwiches and desserts, featuring Rose's vanilla custard. **It's all delicious, but it's not the cheapest meal in town.** Decor is bookstore chic, and Kepler's Books & Magazines is just across the way. On warm evenings you can dine outside in the courtyard. Open Mon.–Fri. 7 a.m.–11 p.m., Sat. 9 a.m.–11 p.m., Sun. 9 a.m.–5 p.m. No credit cards. (Reviewed May 23, 1990)

FIG. 1. Example output from *htmldiff*

Figure 1. The icons (which are in color in the actual output) represent different kinds of change operations: insertions, updates, deletions, etc. For details, see [9].

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 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 [15]. Thus, we again need a method to compute, represent, and query changes in the context of semistructured data.

### 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 [9, 7], 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*), designed at Stanford as part of the *Tsimmis* project [8]. 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 [2].) 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 [2]. 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., [4, 5].

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 did not need to build from scratch yet another database system; instead, we capitalized on an existing database system for semistructured data. Finally, as an important first application of DOEM and Chorel, we describe our design and implementation of a *query subscription service* that permits users to subscribe to changes in semistructured data.

### 1.3. 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 approach is used by temporal databases [21, 22]. 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. We use the snapshot-delta approach in our work. An early, simple example of this approach is the idea of *delta relations*, used in active databases [3, 23] and trigger languages [14], 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 [12, 11]. 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.

A preliminary version of this paper appeared in [6]. That version omitted details on several topics, including the properties of DOEM databases, the encoding and translation schemes for Chorel, and implementation issues. Further, that version did not include the description of virtual annotations and snapshot-based access (covered in Section 6 of this paper.)

### 1.4. 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 by translation, and we briefly describe our system implementation. In Section 6, we introduce some extensions to our language that make snapshot-based access in our data model more convenient. We also describe how our translation-based implementation of Chorel is extended for this purpose. Section 7 describes the query subscription system we have implemented based on the material in Sections 3–5. We conclude in Section 8.

## 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 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 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.** 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 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 2 shows a small portion of the data. 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 an 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 [2]. 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 [2]; the basic change operations defined here reflect the actual changes at the database level.

*Example 2.*

Let us consider some modifications to the OEM database in Example 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 modification 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

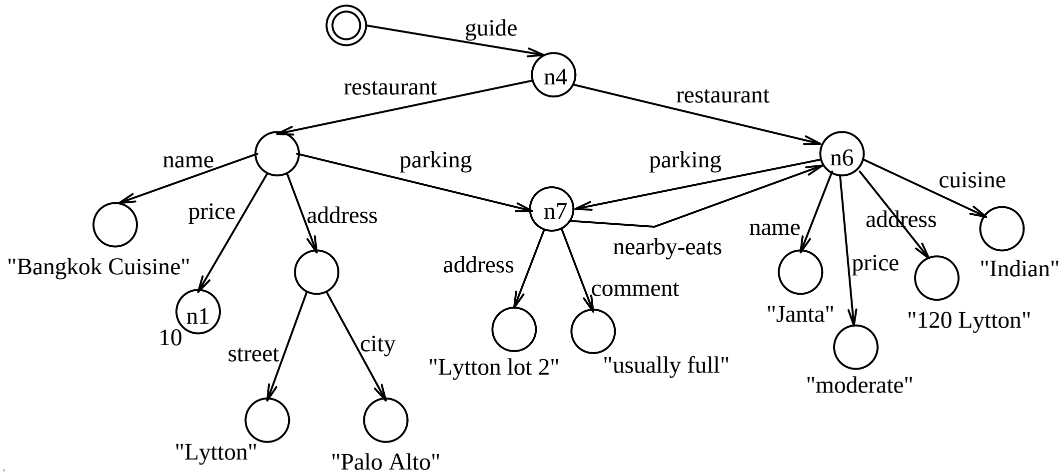



FIG. 2. The OEM database in Example 1.

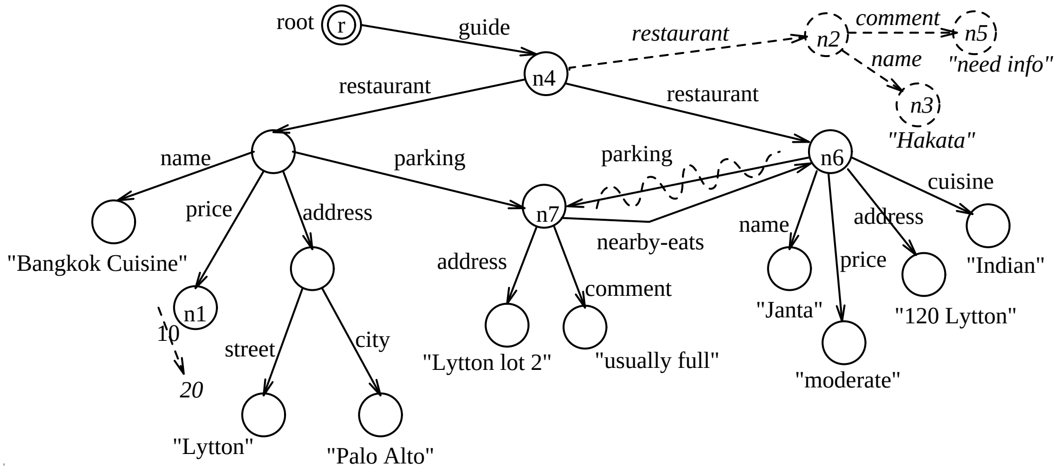



FIG. 3. The OEM database in Example 2

is removed; this modification corresponds to a *remArc* operation. The resulting modified OEM representation of the Guide data is shown in Figure 3, with new data highlighted in bold, and the deleted arc represented using a dashed arrow. □

### 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.**

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 $L_1, ..., L_n$ 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 3.*

The history for the modifications described in Example 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 where $U_i$ are as follows:

$$\begin{aligned} U_1 &= \{\mathit{updNode}(n_1, 20), \mathit{creNode}(n_2, \mathcal{C}), \mathit{creNode}(n_3, \\ &\quad \text{“}\mathit{Hakata}\text{”}), \quad \mathit{addArc}(n_4, \text{“}\mathit{restaurant}\text{”}, n_2), \\ &\quad \mathit{addArc}(n_2, \text{“}\mathit{name}\text{”}, n_3)\} \\ U_2 &= \{\mathit{creNode}(n_5, \text{“}\mathit{need\ info}\text{”})\mathit{addArc}(n_2, \\ &\quad \text{“}\mathit{comment}\text{”}, n_5)\} \\ U_3 &= \{\mathit{remArc}(n_6, \text{“}\mathit{parking}\text{”}, n_7)\}. \end{aligned}$$

This history is valid for the OEM database of Figure 2. □

## 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. We first introduce the annotations we use and define a DOEM database as an OEM graph containing these annotations. Next, we describe how an OEM history (defined in Section 2.2) is represented using a DOEM database. Finally, we discuss some properties of DOEM databases that make them well-suited for representing changes in semistructured data.

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 annotations:

- $\mathit{cre}(t)$: the node was created at time $t$.
- $\mathit{upd}(t, ov)$: the node was updated at time $t$; $ov$ is the old value.
- $\mathit{add}(t)$: the arc was added at time $t$.
- $\mathit{rem}(t)$: 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.**

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 $H = (t_1, U_1), ..., (t_n, U_n)$ 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 $(t_1, U_1), ..., (t_{i-1}, U_{i-1})$, for 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 1.3.

*Example 4.*

Consider the history described in Example 3, which transforms the OEM database of Figure 2 to that of Figure 3. The corresponding DOEM database is shown in Figure 4. 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

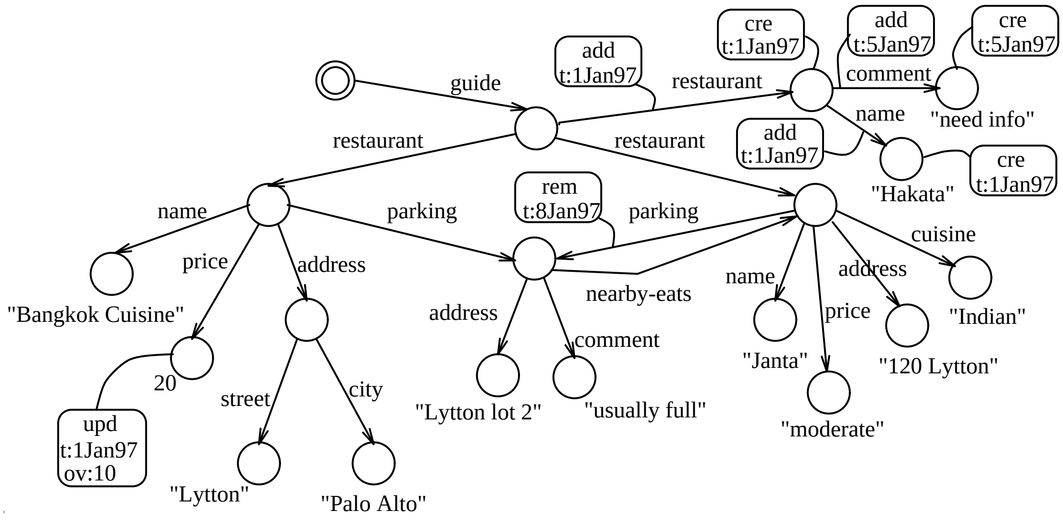



FIG. 4. The DOEM object in Example 4.

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 have seen above how a DOEM database is used to represent an OEM database and its history. We now discuss the advantages of this representation. 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. DOEM databases have the following desirable properties:

- It is easy to obtain the *original snapshot* $O_0(D)$ from a DOEM database $D$. $O_0(D)$ contains exactly those nodes in $D$ that do not have a *cre* annotation. The arcs of $O_0(D)$ are the arcs in $D$ that either have no annotations, or have a *rem* annotation as the annotation with the smallest (earliest) timestamp.
- It is easy to obtain the *snapshot at time t*, $O_t(D)$, from a DOEM database $D$. Starting from the root object of $D$, we traverse $D$ in preorder. For each node $n$ we encounter, we do the following:

  1. We find the value $v_t(n)$ of $n$ at time $t$ (atomic value or $\mathcal{C}$) as follows: If $n$ has no *upd* annotations, then $v_t(n) = v(n)$. Otherwise, let $upd(t_1, ov_1), \ldots, upd(t_k, ov_k)$ be the *upd* annotations in $f_N(n)$. If $t_k \leq t$, $v_t(n) = v(n)$. Otherwise, pick $i \in [1, k]$ such that $t_i$ is the smallest timestamp greater than $t$ in $t_1, \ldots, t_k$; then $v_t(n) = ov_i$.
  2. If $v_t(n) = \mathcal{C}$, continue the preorder traversal by following the arcs emanating from $n$ that were present at time $t$. These are the arcs emanating from $n$ that either do not have any annotation with timestamp less than or equal to $t$, or have an *add* annotation as the annotation with the greatest timestamp less than or equal to $t$.

- It is easy to obtain the *current snapshot* from a DOEM database. It is the *snapshot at time c*, where $c$ is the current time.
- It is easy to obtain the *encoded history* $H(D)$ from a DOEM database $D$. The history $H(D) = (t_1, U_1), ..., (t_n, U_n)$ is constructed as follows. First, $t_1, \ldots, t_n$ is the set of timestamps occurring in $D$, ordered by time. For each $i = 1 \ldots n$, $U_i$ contains the following operations:

  1. $addArc(p, l, c)$ ($remArc(p, l, c)$), if the arc $(p, l, c)$ has the annotation $add(t_i)$ (respectively, $rem(t_i)$);
  2. $updNode(n, v)$, if $n$ has an annotation $upd(t_i, ov)$ and $v$ is the next value of $n$. That is, $v = ov'$ if the next (by time) annotation of $n$ is $upd(t_j, ov')$, and $v = v(n)$ if $n$ is not updated after $t_i$;
  3. $creNode(n, v)$, if $n$ has the annotation $cre(t_i)$, where $v$ is defined as in Case 2.

- It is relatively easy to determine if a given DOEM database $D$ is feasible. We construct the original snapshot $O_0(D)$ and the encoded history $H(D)$ for $D$ as above, and test if $D(O_0(D), H(D)) = D$.
- Most importantly, if $D$ is feasible, 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.

- 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 [2] 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 [5] 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. We do not describe Lorel in detail here (see [2]), but only present through a simple example those features that are needed to understand Chorel.

*Example 5.*
Consider again the OEM database depicted in Figure 3. 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 OEM database shown in Figure 3 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 result is an intuitively reasonable response to the original query, despite the typing difficulties and the missing data. □

Lorel also allows the use of path expressions that include regular expressions and wildcards (e.g., "#" matches an arbitrary path of length 0 or more). Such *general path expressions* are powerful extensions of the simple path expressions of OQL, and allow Lorel users to specify complex path patterns in a database graph. Chorel is also based on extending the notion of path expressions, but in a different direction: We extend path expressions to allow the annotations in DOEM databases to be specified and matched.

### 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 6.*
Consider the DOEM database depicted in Figure 4. 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 4, 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:

$$< Annot[\mathtt{at}\, timeV] > \text{ if } Annot \in \{ \mathtt{add}, \mathtt{rem}, \mathtt{cre} \}$$
$$< \mathtt{upd}[\mathtt{at}\, timeV][\mathtt{from}\, oldV][\mathtt{to}\, newV] > \text{ for } \mathtt{upd}$$

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 as shown in Section 3.2.

Let us consider a Chorel query that uses a time variable. Note that we allow users to enter timestamps using a textual representation, e.g., 4Jan97. In keeping with Lorel's extensive use of coercion, any recognizable format is allowed and is converted automatically to an internal timestamp datatype.

*Example 7.*

Consider the DOEM database in Figure 4. 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 4, 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 feature is illustrated by the following query, which uses time and data variables in the `select` clause.

*Example 8.*

Referring again to the DOEM database in Figure 4, 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 (on the left):

```
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"
   update-time 1Jan97
   new-value 20
```

The result of the above query is a single complex object with three components, as shown on the right. The label *name* is chosen by Chorel using the method described in [2]. 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*. The details of this rewriting are described in [2].

At this stage, we have to give a semantics to range variable definitions that may include annotation expressions (e.g., "`X.label Y`," "`X.<add at T>label Y`") in the context of a DOEM database. In the absence of an annotation expression, the semantics of an expression "`X.label 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 *label* 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 8 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:

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

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 [2]. Consider the following example:

*Example 9.*
Consider again the DOEM database of Figure 4. 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 have implemented DOEM databases and Chorel queries. One approach would be to extend the kernel of the *Lore* database system [13] to allow annotations to be attached to the nodes and arcs of an OEM database. Given these extensions, the Lorel query engine could be extended to a Chorel query engine in a straightforward manner based on the semantics specified in Section 4.3. We do not discuss this approach further. Instead, our implementation uses an alternative approach of implementing DOEM and Chorel "on top of" Lore. 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 the direct implementation approach (and not affecting Lore object layout or query processing), this approach can also be adapted easily to other graph-based data models, e.g., those in [4, 5]. Note that while there are several simple methods of encoding a DOEM database as an OEM database, the challenge here is to devise an encoding that permits a simple and valid translation of Chorel queries over the original DOEM database into Lorel queries over the OEM encoding. For many of the obvious possible encodings, such query translation proves to be very difficult or impossible.

We begin by explaining how we encode DOEM databases in OEM, followed by a description of the translation of Chorel queries to Lorel queries for this encoding, and finally a description of our system implementation.

### 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 $O_D$. Atomic objects are encoded as complex objects so that we can record their histories using subobjects. Special labels used by the encoding start with the character "`&`" to distinguish them from standard labels occurring in $O$. The encoding object $o'$ for DOEM object $o$ has the following subobjects, listed by their labels. Refer to Figures 5 and 6.

- `&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. (The use of this extra edge will soon become clear.)
- `&cre`: If $o$ has a create annotation $cre(t)$, then $o'$ has a "`&cre`"-labeled complex subobject $o'_c$ that has a "`&time`"-labeled atomic subobject with value $t$.
- `&upd`: For each update annotation $upd(t, ov)$ attached to $o$, $o'$ has an "`&upd`"-labeled complex subobject $o'_u$. The object $o'_u$ has a "`&time`"-labeled

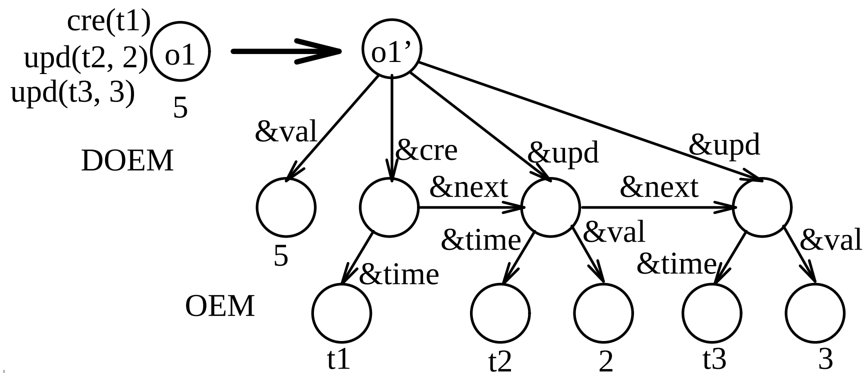



FIG. 5. Encoding a DOEM object in OEM: node annotations

atomic subobject with value $t$, and a "`&val`"-labeled atomic subobject with the value before the update ($ov$).

- $l$: 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 $(o', \&l\texttt{-history}, o'_l)$ 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:
  - `&target`: There is an arc $(o_l, \&\texttt{target}, p')$, where $p'$ is the object encoding $p$.
  - `&add`, `&rem`: For each annotation $add(t)$ ($rem(t)$) attached to $(o, l, p)$, there is an "`&add`"-labeled (respectively, "`&rem`"-labeled) complex subobject $o'_c$ that has a "`&time`"-labeled atomic subobject with value $t$.
- `&next`: For each OEM object $o'_1$ that encodes a DOEM object $o_1$ and its node annotations, the "`&cre`"- and "`&upd`"-labeled subobjects of $o'_1$ are chained together in ascending order of the values of their "`&time`" subobjects using arcs with label "`&next`." (As we shall see shortly, this chaining is useful for obtaining the "new value" corresponding to an update annotation.) Similarly, for each OEM object $o'_{iLj}$ that encodes a DOEM arc $(o_i, L, o_j)$ and the annotations on that arc, the "`&add`"- and "`&rem`"-labeled subobjects of $o_{iLj}$ are chained together in ascending order of the values of their "`&time`" subobjects using arcs with label "`&next`." (As we shall see in Section 6, this chaining is useful for implementing snapshot-based access.)

### 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. In Section 4.3 we described how a Chorel query can be rewritten into an OQL-like query using special functions *creFun*, *updFun*, *addFun*, and *remFun*. Therefore, in the following we assume that we are given such a rewritten query.

We simulate the special functions *creFun*, *updFun*, *addFun*, and *remFun* using expressions that extract the required values from the OEM encoding of the annotations. For example, the expression "`(T, OV, NV) in` *updFun*`(P)`" is replaced with "`P.&upd U, U.&time T, U.&val OV, U.&next.&val NV`." From the encoding scheme described in Section 5.1, we see that this expression instantiates the triple `(T, OV, NV)` to the timestamp, old value, and new value of the update annotations on objects bound to `P`. If an expression of the form "`(T, C) in` *addFun*`(P, l)`" occurs in a Chorel query, we replace it with "`P.&l-history H, H.&add.&time T, H.&target C`." The case for remove annotations, involving the *remFun* function, is analogous. Finally, we replace an expression "`T in` *creFun*`(P)`," with "`P.&cre.&time T`."

Note that our encoding scheme ensures that only arcs that exist in the current snapshot corresponding to the encoded DOEM database are accessible directly via their labels in the encoding. If an $l$-labeled arc does not exist in the current snapshot, its information is stored using an arc with label `&`$l$`-history`, which does not match the label $l$.

One remaining issue is that in the OEM encoding of a DOEM database, the value of an atomic object is stored in a "`&val`"-labeled subobject of the encoding object. So, for instance, when a query compares an atomic object to a value, we want to use the value stored in the "`&val`" subobject for this comparison. Therefore, wherever in the query the value of a object variable is accessed (i.e., in predicates and function arguments) we replace the object variable "`X`" with "`X.&val`." Observe that since there is a "`&val`"-labeled arc from the encoding of each complex object to itself, we can safely perform the above transformation for all value accesses of object variables occuring in the original query, without

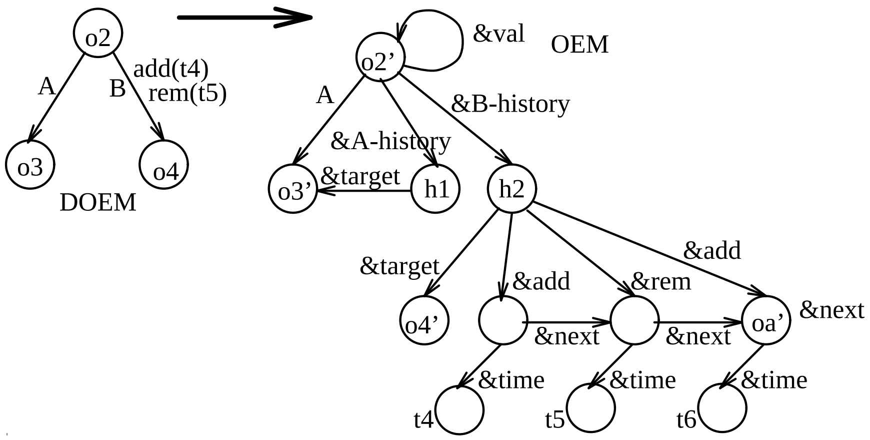



FIG. 6. Encoding a DOEM object in OEM: arc annotations

knowing whether the objects they encode are atomic or complex (which, in general, we will not know).

The transformation is illustrated by the following example.

*Example 10.*

Consider the Chorel query in Example 9. In Section 4.3, we considered the OQL-like rewriting of this query. We now complete this rewriting as described above, to yield the following Lorel query over the OEM encoding of the DOEM database in Figure 4:

```
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.&time :
      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 example 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 the user to access both the value and the history of an object.

In the above description, for simplicity we assumed that every atomic object $o$ is encoded using a complex object $o'$ that has a &val-labeled subobject with value $v(o)$. However, in practice we do not encode unannotated atomic objects in this manner; that is, if an atomic object $o$ has no annotations, we encode it using a simple atomic object $o'$ with value $v(o)$. In our translation scheme, we replace accesses to the value of an variable X by X.[&val], which is a Lorel path expression indicating an optional path component &val.

## 6. Virtual Annotations and Snapshot-based Access

In Section 4.2 we have seen how the construct <upd at T from oldV to newV> refers to a *virtual annotation* $upd(t, ov, nv)$, where $t$, $ov$, and $nv$ are, respectively, the timestamp, the old value, and the new value of an update operation in the history. The real annotation, $upd(t, ov)$, does not contain the old value, however that information is available elsewhere in the database. We can extend this idea of virtual annotations to facilitate access to other implicit information in a DOEM database. As a concrete example, in this section we introduce virtual annotations that facilitate *snapshot-based* access to a DOEM database. (Recall Section 1.3, which describes different modes of accessing historical information.) We define the semantics of Chorel queries containing references to virtual annotations by using range functions that are defined over the real annotations and data in a DOEM database. We describe how to implement this added functionality by extending the translation-based method of Section 5.

### 6.1. *Snapshot-based Access*

Recall from Section 4.3 that an unannotated path expression such as guide.restaurant.entree.price is evaluated over the *current snapshot* of a DOEM database. Sometimes, one may wish to evaluate path expression components over other (non-current) snapshots. For example, we may wish to refer to the price of an entree at some time $T$; we introduce the syntax guide.restaurant.entree.price<at T>. Similarly, we may wish to refer to the existence of a *parking* arc between two objects $X$ and $Y$ at time $T$; we use the syntax X.<at T>parking Y in the from clause of a Chorel query.

*Example 11.* Consider the Guide database depicted in Figure 4. Suppose we wish to list the parking areas close to the restaurant "Janta" as of 1st January 1997. We write the following query:

```
select P
from guide.restaurant R, R.<at T>parking P
where R.name = "Janta" and T = 1Jan97;
```

For the DOEM database depicted in Figure 4, this query returns the parking object with address "Lytton lot 2," since on 1st January 1997 there was a "parking" arc from the Janta restaurant object to the Lytton parking object. (This arc was removed on 8th January 1997.) □

When the variable $T$ occuring in an `at` annotation expression is bound to a constant elsewhere in the query (as in the above example), the effect of the annotation expression on query evaluation is intuitively simple: We evaluate the query as if the path expression component qualified by `<at T>` refers to the snapshot of the database at time $T$. As we have seen in Section 3.2, the snapshot at time $T$ is easily obtained using the information in a DOEM database. However, if $T$ is unbound, then unless we take special precautions we may find ourselves faced with unsafe queries, as illustrated by the following example.

*Example 12.* For the Guide database depicted in Figure 4, suppose we are interested in finding the times at which the restaurant "Bangkok Cuisine" had a price rating less than 15. We write the query as follows:

```
select T
from guide.restaurant R, R.price<at T> P
where R.name = "Bangkok Cuisine" and P < 15;
```

The basic problem with this query is that while the database stores only a finite number of timestamps, the above query would require $T$ to range over the infinite number of intermediate timestamp values as well. □

We overcome such difficulties by allowing timestamp variables such as $T$ above to bind only to those timestamp values that exist explicitly in the DOEM database. Intermediate timestamp values are represented using intervals $[B, E)$, where $B$ and $E$ are the begin and end timestamps, respectively. (We use a convention of intervals that are closed on the left and open on the right; our methods are not dependent on this convention.)

To introduce this concept of intervals, we add another virtual annotation, called *during*, on nodes and arcs, and a corresponding annotation expression "`<during B E>`" in the syntax of annotated path expressions. (As we will see in Section 6.3, virtual annotation *during* in fact subsumes virtual annotation *at*.) Intuitively, the construct `X<during B E> V` in a `from` clause binds the triple $(B, E, V)$ to all values $\{(b, e, v)\}$ such that the object $X$ had value $v$ continuously from time $b$ to time $e$. Similarly, the construct `X<during B E>l Y` binds the triple $(B, E, Y)$ to all values $\{(b, e, Y)\}$ such that the arc $(X, l, Y)$ existed continuously from time $b$ to time $e$. We further require that the above intervals $[b, e)$ be maximal.

When using snapshot-based access, we often need to refer to the current time. We introduce a distinguished timestamp $t_N$ for this purpose. More precisely, $t_N$ is a special variable whose value during the evaluation of a query is the time at which that evaluation begins. Similarly, we often need to refer to the initial timestamp corresponding to a database; we introduce a distinguished timestamp $t_I$ for this purpose. More precisely, each DOEM database has an initial timestamp $t_I$ associated with it. Note that $t_I$ is a constant, and may be negative infinity.

Using the `during` virtual annotation, the query in Example 12 may be rewritten as follows:

```
select B,E
from guide.restaurant R,
   R.price<during B E> P
where R.name = "Bangkok Cuisine" and P < 15;
```

This query returns a set of pairs $\{(b, e)\}$ such that at all times during the interval $[b, e)$, Bangkok Cuisine had a price rating less than 15. For our example database depicted in Figure 4, this query returns the singleton set $\{(t_I, 1Jan97)\}$, where $t_I$ is the initial timestamp of the database.

Note that it *is* possible to express such snapshot-based queries using only the basic Chorel constructs described in Section 4. However, the resulting queries are extremely cumbersome. For example, the simple snapshot-based access `X.<during B E>foo Y` in a *from* clause requires a construction such as the following:

```
from X.<add at B>foo Y, X.<rem at E>foo Z...
where Y = Z and not exists M :
   (X.<add at M>foo Y or X.<rem at M>foo Y);
```

In reality, the expression is even more complex, since we need to handle the special cases involving missing annotations on both the "begin" and the "end" side. Thus, snapshot-based access is an excellent candidate for simplification using virtual annotations.

### 6.2. Semantics of `during`

We now formalize our intuitive description of the semantics of `during` annotations. As in Section 4.3, we shall specify the semantics using a rewriting with special functions for binding range variables. To define the semantics of the arc annotation expression `X.<during B E>l Y` in the `from` clause of a Chorel query, we introduce a special function, *arcDuring*. This function maps a DOEM object $o_1$ and label $l$ to a set of triples $\{(b, e, o_2)\}$ such that in the history represented by the DOEM database, the arc $(o_1, l, o_2)$ existed in the time interval $[b, e)$, and $[b, e)$ is maximal (i.e., this condition fails to hold if we decrease $b$ or increase $e$). We rewrite the `from` clause by replacing `X.<during B E>l Y` with `(B,E,Y) in` *arcDuring*`(X,l)`. (Recall from Section 3.2 that given a DOEM database $D$, it is easy to obtain the snapshot at time $t$, $O_t(D)$. Thus the intervals $[b, e)$ in the definition of *arcDuring* are well defined.) The function *arcDuring* has some notable boundary cases: If the earliest annotation on an arc is $rem(t_1)$, then the arc exists in $[t_I, t_1)$. (Recall from Section 6.1 that $t_I$ is the initial timestamp associated with a database and $t_N$ is the current timestamp.) Similarly, if the latest annotation on an arc is $add(t_2)$, then the arc exists in the interval $[t_2, t_N]$. Finally, if an arc has no annotations, it exists in $[t_I, t_N]$.

Now we define the semantics of the node annotation expression `X<during B E> V` in the `from` clause of a Chorel query. To do so, we introduce a special function, *nodeDuring*. This function maps a DOEM object $o$ to a set of triples $\{(b, e, v)\}$ such that in the history represented by the DOEM database, the object $o$ had value $v$ during the time interval $[b, e)$, and $[b, e)$ is maximal. We rewrite the `from` clause replacing `X<during B E> V` with `(B,E,V) in` *nodeDuring*`(X)`. (Using Section 3.2 we see that the intervals $[b, e)$, and the corresponding values $v$, are well-defined.) The function *nodeDuring* also has some notable boundary cases: If the earliest annotation on a DOEM object $o$ is $upd(t_1, v_1)$ then $o$ has value $v_1$ in the interval $[t_I, t_1)$. Similarly, if the latest annotation on $o$ is $upd(t_k, v_k)$, then $o$ has value $v(o)$ (the current value) in $[t_k, t_N]$. Finally, if $o$ has no annotations, then it has value $v(o)$ in $[t_I, t_N]$.

*Example 13.* Consider the query proposed in Example 11. Using the `during` construct, we can write the following query to return parking for the "Janta" restaurant as of 1st January 1997.

```
select P
from guide.restaurant R,
   R.<during B E>parking P
where R.name = "Janta" and B <= 1Jan97
   and E > 1Jan97;
```

Using the semantics for `during` described above, we see that this query is conceptually rewritten to the following:

```
select P
from guide.restaurant R,
   (B,E,P) in arcDuring(R,parking)
where R.name = "Janta" and B <= 1Jan97
   and E > 1Jan97;
```

Consider the DOEM database in Figure 4. When $R$ is bound to the restaurant object "Janta," function *arcDuring* results in the tuple variable $(B, E, P)$ ranging over the singleton set $\{(t_I, 8Jan97, p_1)\}$, where $p_1$ is the parking object with address "Lytton lot 2." Since $R$, $B$, and $E$ satisfy the predicate in the `where` clause, the Lytton parking object will be returned as the query result. □

### 6.3. The `at` Construct

Examples 11 and 13 suggest a simple definition for the edge annotation `X.<at T>l Y` and the node annotation `X<at T> V`. We define them as abbreviations for `X.<during B E>l Y` and `X<during B E> V`, respectively, and add the condition `B <= T < E` to the `where` clause. Note that our rewriting requires the variable `T` occurring in the `at` annotation to be bound elsewhere in the query independently of the path expression component containing `at`. For example, if we apply this definition of `<at T>` to rewrite the query in Example 11, we obtain the query in Example 13.

In cases where the variable `T` occurring in the `<at T>` construct is not bound elsewhere in the query, the definition of `at` as an abbreviation for a `during` expression fails. For example, if we apply the rewriting to the problematic query of Example 12, which uses `<at T>` without binding `T` elsewhere, we get the following query in which `T` is still unbound:

```
select T
from guide.restaurant R,
   R.price<during B E> P
where R.name = "Bangkok Cuisine" and P < 15
   and B <= T and T < E;
```

In general, this problem can be mitigated by allowing timestamp variables such as `T` to bind to intervals instead of single timestamps. However, we do not consider such extensions further in this paper. We shall henceforth assume that the `<at T>` construct is defined only when `T` is bound elsewhere in the query independently of the path expression component containing `at`.

### 6.4. The `snap` Construct

Let us now consider a special class of Chorel queries that are useful in studying past states of a historical database. Intuitively, such queries take the snapshot at some time $t$, and then evaluate an ordinary (non-historical) query over this snapshot. We call such queries *pure snapshot queries*. For example, using our Guide database, suppose we wish to generate, as of 15th June 1997, the names, price ratings, and parking addresses for restaurants with a price rating less than 20. That is, we would like to evaluate the following Lorel (non-historical) query over the OEM database that is the DOEM snapshot of 15th June 1997:

```
select R, P, A
from guide.restaurant R, R.price P,
   R.parking.address A
where R.price < 20;
```

In reality we are evaluating Chorel queries over our DOEM database. Thus, to express that the above query should be evaluated over the snapshot of 15th June 1997, we could qualify each component of each path expression in the query as follows:

```
select R, P, A
from guide.<at T>restaurant R,
   R.<at T>price<at T> P,
   R.<at T>parking.<at T>address<at T> A
where R.<at T>price<at T> < 20
   and T = 15Jun97;
```

In order to make writing such *snapshot queries* more convenient, we introduce as a syntactic convenience the construct <snap T>, with the requirement that T be bound elsewhere in the query independently of the path expression component containing snap. The construct X.<snap T>foo Y in a from clause is rewritten to X.<at T>foo Y; furthermore, any other use of Y in the query is (recursively) rewritten as though it were qualified by a <snap T>. In particular, Y.bar Z is interpreted as Y.<snap T>bar Z and recursively rewritten, and accesses to Y's value are rewritten as Y<at T>. The where clause is handled analogously. Using this construct, the above query may now be written more simply as follows:

```
select R, P, A
from guide.<snap T>restaurant R, R.price P,
   R.parking.address A
where R.price < 20 and T = 15Jun97;
```

### 6.5. *Implementing* during *by translation*

We now describe how the translation-based implementation of Chorel described in Section 5 is extended to accommodate the during construct. Refer to Figures 7 and 8, which depict the OEM encoding of DOEM objects; we have indicated the new features using dashed lines. (The other features were described in Section 5.1.)

Each OEM database used to encode a DOEM database has a special complex object $o'_N$ that has one "&time"-labeled atomic subobject $o''_N$ with value $t_N$. (Recall, from Section 6.1, that $t_N$ refers to the current time; in the implementation, the value of $o''_N$ is the query execution time.) Similarly, there is a special complex object $o'_I$ that has one "&time"-labeled atomic subobject $o''_I$ with value $t_I$. (Recall, from Section 6.1, that $t_I$ is the initial timestamp associated with a DOEM database, and may be negative infinity.) Note that there is exactly one instance of each of the objects $o'_N$, $o''_N$, $o'_I$, and $o''_I$ per database. (To highlight this fact, these objects are depicted using shaded circles in Figures 5 and 6.)

In Section 5.1, we described the use of "&next"-labeled arcs to chain annotation-encoding objects in ascending order of the annotation timestamps. We now extend this chain to include the timestamps $t_I$ and $t_N$ as follows. Consider first the encoding of node annotations, as depicted in Figure 5. If a DOEM node $o$ has one or more node annotations (create or update), then in its OEM encoding, we add a "&next"-labeled arc from the object encoding the annotation with the largest timestamp to the special object $o'_N$. The "&next"-labeled arc from $o'_u$ to $o'_N$ in Figure 5 is an example of this case. If the DOEM node $o$ has no annotations, then in the OEM encoding, we add a "&dcre"-labeled arc from the corresponding node $o'$ to the special node $o'_I$. In Figure 5, if $o_1$ were to not have a create annotation, a "&dcre"-labeled arc from $o'_1$ to $o'_I$ would exist. (Since in reality $o_1$ does have a create annotation, this "&dcre"-labeled arc does not exist, and is depicted using a dotted line.)

Now consider the encoding of arc annotations, as depicted in Figure 6. If an arc $(o_1, l, o_2)$ in the DOEM database has no annotations, then in the OEM encoding of the database, we add a "&dadd"-labeled arc from $o'_{1l2}$ to the special object $o'_I$, where $o'_{1l2}$ is the "&$l$-history"-labeled subobject of $o'_1$ that encodes the history of $(o_1, l, o_2)$. In Figure 6, $o_{1l2}$ is shown as the object $h_1$. If the arc $(o_1, l, o_2)$ has one or more annotations, and the annotation with the largest timestamp is an *add* annotation, then the OEM encoding has a "&next"-labeled arc from the corresponding "&add"-labeled subobject $o'_a$ of $o'_{1l2}$ to the special object $o'_N$. In Figure 6, we see an example of such an arc from $o'_a$ to $o'_N$.

Given the above enhancements to our scheme for encoding DOEM in OEM, we can rewrite Chorel queries containing the during construct as Lorel queries over the encoding objects. Given a Chorel query with the construct X<during B E> in the from clause, we replace this construct by the following: X(.&cre|.&upd|.&dcre) A, A.&time B, A.&next.&time E, A.&next.&val V. Similarly, if a Chorel query has the construct X.<during B E>foo Y in the from clause, we replace this construct by the following: X.&foo-history H, H.&target Y, H(.&add|.&dadd) A, A.&time B, A.&next.&time E. As in Section 5, variables introduced in the where clause of a Chorel query are treated by introducing existential quantification in the where clause.

*Example 14.* Consider the during-based query in Example 13. Using the above rewriting, we obtain the

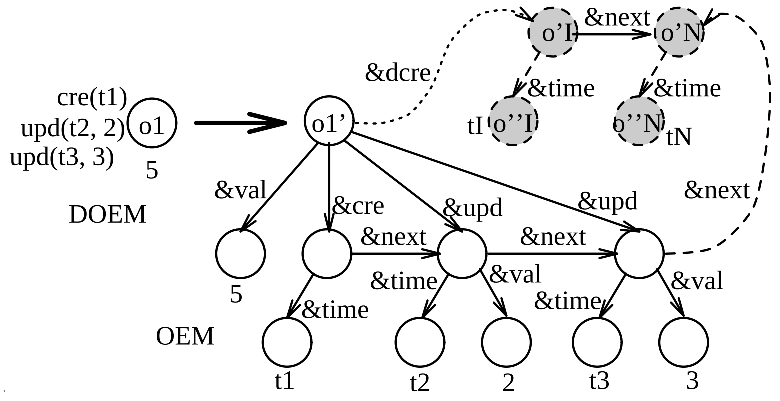



FIG. 7. Encoding a DOEM object in OEM: node annotations

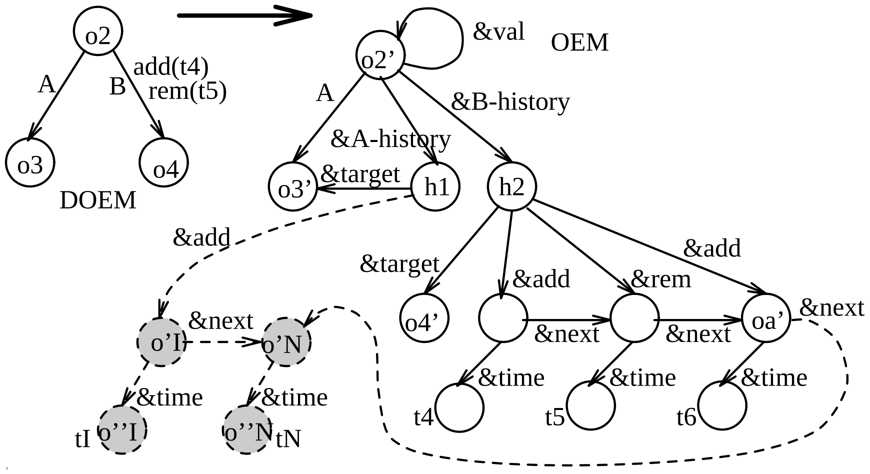



FIG. 8. Encoding a DOEM object in OEM: arc annotations

following Lorel query over the OEM database encoding the Guide DOEM database:

```
select P
from guide.restaurant R,
   R.&parking-history H, H.&target P,
   H.&add A, A.&time B, A.&next.&time E
where R.name = "Janta" and B <= 1Jan97
   and E > 1Jan97;
```

□

### 6.6. *Object Deletion and Garbage Collection*

Recall that in the OEM data model underlying DOEM and Chorel, there is no explicit object deletion operation. Instead, persistence is by reachability from the distinguished root of the database, and any unreachable objects are implicitly deleted. An OEM database system must therefore periodically perform garbage collection in order to detect and remove such deleted objects. Between the time an object becomes unreachable and the time garbage collection is performed, the semantically deleted object continues to exist in the database. This situation does not pose any difficulties for Lorel queries, since Lorel path expressions cannot access any object that is unreachable from the root of the current database snapshot. However, in Chorel, such deleted objects are reachable using annotated path expressions that contain a "forward jump in time" (i.e., path expressions that refer to a more recent snapshot from an older one). The following example illustrates the point:

*Example 15.* Referring back to our Guide database depicted in Figure 4, suppose the arc from the Guide object to the restaurant object for "Bangkok Cuisine" is removed on 1st July 1997. This arc removal results in the restaurant object for Bangkok Cuisine, as well as its price, address, street, and city subobjects becoming unreachable from the root of the database, implying their deletion. In our DOEM database, however, these objects continue to exist; the only change is that there is now a remove annotation $rem(1Jul97)$

on the restaurant arc that was removed. Now suppose on 15th July 1997 we issue the following query to our DOEM database, asking for the current price rating of all restaurants that existed as of 1st June 1997:

```
select P
from guide.<at 1Jun97>restaurant R,
   R.price P;
```

Now since the *price* object for Bangkok Cuisine does not currently exist, the result of the above query should not contain it. However, there is no way for the Chorel query engine to detect this situation, since there is no information on either the *restaurant* or the *price* objects that suggests their deletion. (The relevant piece of information is the *rem* annotation on the restaurant arc.) Thus the query result will contain the price rating for Bangkok Cuisine. □

We mitigate the above problem by introducing a *delete annotation*, which records the deletion of an object (usually as a result of garbage collection). Suppose that at time $t_G$, some objects are determined to be newly unreachable from the root of the database. In the corresponding DOEM database, we mark such newly unreachable objects (which continue to exist physically) using a $del(t_G)$ annotation. We further ensure that we do not access the value of an object at time $t'$ if that object has a $del(t)$ annotation with $t' > t$. More precisely, we modify the definition of the *nodeDuring* function in Section 6.2 to state that if a node has a $del(t_d)$ annotation then its value after $t_d$ is undefined. (That is, the most recent time interval is modified from $[t_k, t_N]$ to $[t_k, t_d)$.) The corresponding changes to the translation-based implementation are straightforward.

## 7. A Query Subscription Service

In Section 1, we mentioned as 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 our 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 are 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* [20] 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* [17, 16], 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 [9, 7] 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; refer to Figure 9. The first component is a pair of *frequency specifications* $(f_p, f_f)$. The *polling frequency* $f_p$ indicates the times at which data source is to be polled in order to detect changes. The *filter frequency* $f_f$ indicates the times at which new changes should be evaluated and reported to the user. Examples of frequency specifications are "every Friday at 5:00pm" and "every 10 minutes." The polling frequency implies a sequence of time instants $(t_1, t_2, t_3, \ldots)$, which we call *polling times*. *Filter times* are defined analogously. (In the actual system, we also consider two other modes: one in which the polling and/or filter times are obtained following explicit user requests, and the other in which they 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 describe 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. (Recall from Section 4.1 that "#" is a special character that matches any sequence of zero or more labels in a path. We also use the Lorel operator `like` for string matching.)

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

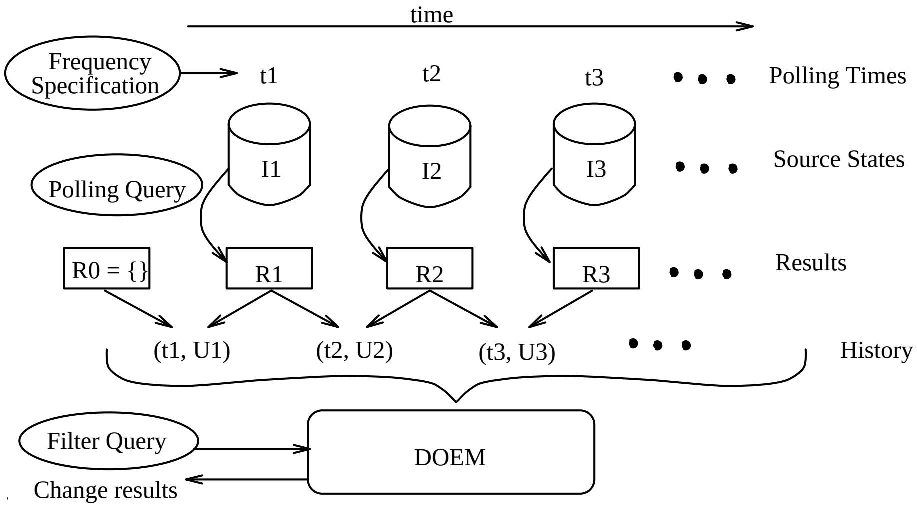



FIG. 9. A Query Subscription Service based on DOEM and Chorel

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 [9, 7], 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 time-step the results of the polling query are all "created." Thereafter, each subsequent time-step annotates the DOEM database with the changes to the result of the polling query since the previous time-step. 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 "`LyttonRests`."

The third component of a subscription is a Chorel query $Q_c$, called the *filter query*, over the generated DOEM database. In addition to standard Chorel, in $Q_c$ we can use a special time variable "`t[0]`" to refer to the current filter time $t_k$, and we can use "`t[-1]`," "`t[-2]`," etc., to refer to the past filter times $t_{k-1}, t_{k-2}$, etc. (If the current filter time is $t_k$, we define `t[-i]` to be $t_{k-i}$ if $i < k$, and $t_I$ otherwise, where $t_I$ is the initial timestamp associated with the DOEM database of the subscription.) The filter query describes the data and changes of interest to the user. An example filter query is the following:

```
define filter query NewOnLytton as
select R.name, C.name
from LyttonRests.restaurant<cre at T1> R,
   LyttonRests.cafe<cre at T2> C
   where R.parking = C.parking and T1 > t[-1]
      and T2 >= 1Jan97;
```

Given our definition of the DOEM database "`Lytton-Restaurants`," this query indicates that the user should be notified of the names of restaurant-cafe pairs on Lytton street that share a parking area, where the restaurant was newly created since the last filter time and the cafe was created some time after January 1, 1997. At each filter time $t_k$ ($k > 0$) given by the filter frequency, QSS evaluates $Q_c$ over the DOEM database $D(R_0, H_k)$, where $H_k = (t_1, U_1), \ldots, (t_j, U_j)$, and $t_j$ is the greatest polling time less than $t_k$, and returns the results to the user.

*Example 16.*

Consider again the changes to the Guide data described in Example 2, as depicted in Figure 3. 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 $t_1 = \mathit{30Dec96}$, $t_2 = \mathit{31Dec96}$, $t_3 = \mathit{1Jan97}$, 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 restaurant objects in Figure 2. 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 $t_2$, $\texttt{t}[-1] = t_1$, 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. 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, $\texttt{t}[-1] = t_2$, 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. □

For certain polling queries, QSS may need to store a large portion of the underlying database in order to detect changes accurately. We are exploring the following ways of limiting the space used for storing DOEM databases: (1) merging the DOEM databases for several subscriptions that have similar polling queries; (2) making the client responsible for storing the DOEM databases for its subscriptions; and (3) trading accuracy for space by storing a smaller state at the expense of not being able to detect all changes accurately. We are also working on methods for determining a polling query and filter query automatically from a simpler form of subscription query.

## 8. 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. We have described our implementation of CORE, a change object repository based on DOEM and Chorel. We have demonstrated how we can use virtual annotations to facilitate snapshot-based access to a historical database. 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 Lorel's wildcards and regular expressions in path expressions [2]. (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 further uses of *virtual annotations*, and alternatives to 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.

## Acknowledgments

We are grateful to Vineet Gossain and Dan Liu for their substantial efforts in implementing CORE and QSS, and to many members of the Stanford Database Group and the Lore and $C^3$ projects for fruitful discussions about change management.

## References

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

[2] 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.

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

[4] 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.

[5] R. Cattell. *The Object Database Standard: ODMG-93 Release 1.2.* Morgan Kaufmann Publishers, San Francisco, California, 1996.

[6] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the International Conference on Data Engineering*, pages 4–13, Orlando, Florida, February 1998.

[7] S. Chawathe and H. Garcia-Molina. An expressive model for comparing tree-structured data. Technical report, Stanford University Database Group, 1997. Available at `http://www-db.stanford.edu/`.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[14] J. Melton. An SQL3 snapshot. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 666–672, New Orleans, Louisiana, February 1996.

[15] 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.

[16] 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.

[17] 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.

[18] The Palo Alto Weekly online, 1998. `http://www.service.com/PAW/`.

[19] 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.

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

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

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

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