

Sync your Data: Update Propagation for Heterogeneous Protein Databases

Kajal T. Claypool¹, and Elke A. Rundensteiner²

¹ Department of Computer Science, University of Massachusetts, Lowell, MA - 01854

² Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA - 01602

Received: date / Revised version: date

Abstract The traditional model of bench (wet) chemistry in many life sciences domain is today actively complimented by computer-based discoveries utilizing the growing number of online data sources. A typical *computer-based discovery* scenario for many life scientists includes the creation of local caches of pertinent information from multiple online resources such as Swissprot [BA00], PIR [BGH⁺00], PDB [PDB03], to enable efficient data analysis. This local caching of data, however, exposes their research and eventual results to the problems of data staleness, that is, cached data may quickly be obsolete or incorrect, dependent on the updates that are made to the source data. This represents a significant challenge to the scientific community, forcing scientists to be continuously aware of the frequent changes made to public data sources, and more importantly aware of the potential effects on their own derived data sets during the course of their research.

To address this significant challenge, in this paper we present an approach for handling update propagation between heterogeneous databases, guaranteeing data freshness for scientists irrespective of their choice of data source and its underlying data model or interface. We propose a *middle-layer* based solution wherein first the change in the online data source is translated to a sequence of changes in the middle-layer; next each change in the middle-layer is propagated through an algebraic representation of the translation between the source and the target; and finally the net-change is translated to a set of changes that are then applied to the local cache. In this paper, we present our algebraic model that represents the mapping of the online resource to the local cache, as well as our adaptive propagation algorithm that can incrementally propagate both schema and data changes from the source to the cache in a data model independent manner. We present a case study based on a joint ongoing project with our collaborators in the Chemistry department at UMass-Lowell to explicate our approach.

1 Introduction

Data Freshness - A Key Issue for Life Sciences Data

Life scientists in this post genomics age, expect to compliment their bench-top (wet Chemistry) research with computing based discoveries. To support this endeavor, over the past decade, scientists have built large repositories of protein databases that make it possible for them to move their current research methodology to a more computer assisted approach. Today, these data sources collectively store the sequences of over 1 million proteins and the detailed atomic structures of over 24,000 proteins. Examples of such data sources include some primary resources that contain only data gathered on one specific organism (GDB [GDB] on the Human Genome Project), others that collect data on all biologically interesting concepts (Uniprot [Uni02] and SWISS-PROT [BA00] on proteins for all organisms), as well as powerful multi-database search engines (SRS [EUA96], TISSMIS [GMHI⁺95], TAMBIS [BBB⁺98], and DiscoveryLink [HKR⁺00]).

A typical working scenario for many life scientists is to create local caches of the pertinent information from several of the relevant data sources. This can be highlighted by a very specific example from our collaborators' (Dr. McDonald and Dr. Vasudevan in Chemistry Department of UMass-Lowell) research. Their research focuses on statistical analysis of amino acids essential to heme incorporation. This is a molecular phylogenetic approach to the apparent conservation of heme contact residues in both primary sequence and, presumably, tertiary (heme pocket) topology. Current on-line databases and retrieval tools are used to evaluate the sequence conservation of the α/β peripheral heme group contacts. This is a formidable problem since the identified sequences for over 700 globin-like structures are stored across multiple data sources. To facilitate this investigation, a local data subset was created by mining data from Swissprot [BA00], a primary data source containing sequences. The subset was restricted to (1) sequences of similar lengths (141 amino acids); (2) the sequences of only the major hemoglobin subunit components in organism's erythrocyte; and (3) only those organisms that assemble a tetrameric ($\alpha_2\beta_2$) protein.

Of course, the *derived* (locally cached) data used in this kind of research becomes outdated quickly and must be refreshed from the original sources. For example in the investigation above, earlier results had yielded 152 hits. More recent searches of Uniprot [Uni02] (a combination of three primary databases: Swissprot [BA00], EBI [EBI02] and PIR [BGH⁺00]) have now yielded 177 hits, indicating a significant increase in available data. Scientists using cached data must thus be aware of the frequent changes that are made to the primary data sources, and the effects of these updates on their own findings and results! This is an enormous task given the growth rate of protein databases. For example, the Swissprot release (45.5) in January 2005, contained 3392 more entries (2%) than release 45, less than three months earlier. The sequence data of 702 existing entries had also been updated and annotations of 45,229 entries had been revised [SIB04].

Updates for many of these data sources are available either in a *non-batch* mode or as a complete set, a *batch*. In the non-batch mode users can query the data

sources and examine the *timestamp* on an entry and manually decide whether or not there has been an update. Note, however, that this update on an entry could be on the sequence, on a comment, on the protein function or on a literature reference. Users must examine each record manually or run a time-consuming difference computation algorithm to extract full update information. In the case of the example above this would involve manually examining 177 records on a monthly basis! Some data sources such as Swissprot [BA00] also provide *batch* updates, that is an update set with every new *release*, containing new as well as modified data records. These updated data sets are typically available for download in either a flat-file format (Swissprot [BA00]), as relational tables (PIR [BGH⁺00]), or more recently in XML format (Swissprot [BA00]). In addition, while the volume of schema changes in these databases is not as high as the data changes, schema or format changes still occur at a steady rate. For example, Swissprot [SIB04] in its last two consecutive releases made structural changes to its keyword file as well as to its comment format.

Manual propagation of updates from primary protein databases to local caches of data are further complicated by the data model diversity that exists in the life sciences database domain. In a survey conducted in 2000-2001 [BK02] of the 111 databases surveyed [BK02], 40% (about 44 databases) of the databases were implemented as flat files, 37% were implemented using the relational model, 6% using the object model and about 3% using the object-relational model. With the popularity of the XML, some of these data sources (3% to 6%) now also publish their data in the XML format. Furthermore, there are several research projects underway to define new data models that seek to provide better description and querying of the life sciences data. Given the increasing number of protein data sources currently on-line (somewhere between 500 and 1000 [BK02]), and this diversity of data models, a manual update propagation approach (batch or non-batch mode) is inevitably tedious, error-prone and consequently obsolete, leaving data under-exploited and under-utilized.

1.1 Our Approach

In our work we now provide an *adaptable* solution to handle the propagation of source changes to the target across data model boundaries - an essential component for handling updates in the life sciences domain. Our work is based on the hypothesis that an adaptable solution to change propagation across data model boundaries must necessarily loosen the tight coupling of the change and the mapping between the source and the target. We thus propose a *middle-layer* solution wherein first the change in the local database is translated to a sequence of changes in the middle-layer; next each change in the middle-layer is propagated through an algebraic representation of the translation between the source and the target; and finally the net-change is translated to a set of changes that are then applied to the target.

The core foundation of our solution is *Sangam* [CR03b, CR03a, CRZ⁺01], a *transformation environment* that provides two key features – the *schema* model and

the *transformation* model. The *schema* model is a graph model that is loosely based on the XML Schema model and can represent schemas from the XML, relational and object models¹. The *transformation* model provides uniform representation of translations between two graphs. The goal of the transformation model is two-fold: first, to provide a transformation language in which data is not constrained to be written in one language, but where the specified graph restructuring (and the transformation of the data they give structure to) can be mapped to several languages; and second to provide ease of reasoning on the correctness of propagations over the well-defined graph restructurings. In our work, we focus on *linear* graph transformations as (1) they are among the most elementary structures from which more complex abstractions can be built. In the life sciences domain linear transformations cover a broad class of queries ranging from simple *select/project* queries over a single source, to more complex *join* queries over multiple data sources²; (2) they represent the common set of transformations that can be mapped to the restructuring capabilities available in languages such as XQuery [W3C01]; and (3) they are proven to be complete. That is, there exists a set of linear operators that can reduce any first-order graph into another first-order graph [GY98].

While, XML and XQuery can be used for the data model and transformations respectively, the complexity of XQuery in particular makes it impractical for application in the life sciences domain. Instead, our work takes a complimentary approach that uses graph based formalism to represent linear transformations. The linear transformations can then be mapped to XQuery or SQL to perform the actual data transformations, thereby reaping the benefits of the query optimizations offered by such languages.

To facilitate the adaptive propagation, we (1) provide a well-defined set of primitive data and schema update operations for the data model in the middle-layer; (2) show how local changes in the relational or the XML model can be mapped to the primitive update operations; and (3) develop an *adaptive propagation algorithm* that incrementally pushes the source change through the transformation to the target. A preliminary version of our adaptive propagation algorithm appeared in [CR04]. A unique feature of the *adaptive propagation algorithm* that fundamentally sets it apart from past view maintenance work on SQL [Kel82, GB95] and SchemaSQL views [KR02a], is its handling of in situ structural changes to the modeled transformations during the propagation process. Operators in modeled transformations (or mappings), by their very nature, exist at the finer granularity of an individual node or edge, unlike SQL or SchemaSQL views where the algebra operators operate on entire sets of data. A change such as the deletion of an attribute, may thus result in the deletion of the node modeling its mapping therein causing an update to the overall structure of the modeled transformation. This is in contrast to view maintenance algorithms that have been presented for SQL views [Kel82, GB95] or SchemaSQL views [KR02a] where changes are made in-place on the individual algebra operators with no effect on

¹ Inheritance in object models is handled via flattening of the hierarchy.

² Linear transformations do not cover queries that map data to schema elements and vice versa. We have found that such queries are, for the most part, not common in the life sciences domain.

the overall structure of the algebra tree. The *adaptive propagation algorithm* must thus now also take into account the effect of the change on the overall structure of the modeled transformation. To address this, in this paper we present a unique 2-pass *adaptive propagation algorithm* that adapts the modeled transformation to any structural changes that may occur as a side effect of the update propagation process.

RoadMap: The rest of the paper is organized as follows. Section 2 gives an overview of our approach. Sections 3- 5 describe our core building blocks for achieving the update propagation across different data models. In Section 3 we present the Sangam model - the core of our middle layer approach that includes both the data model and the transformation model. In Section 4 we present a taxonomy of change operations that can be applied to this data model. In Section 5 we present our core propagation strategy. In Section 6 we present a concrete and comprehensive example to explicate our approach and show how updates can be handled in our system. Related work is presented in Section 7 and we conclude in Section 8.

2 Overview

Figure 1 gives an architectural overview of the connections between the different components that comprise the *Sangam* system. Below we describe the key steps that must be undertaken to accomplish the restructuring of the the public data source or query results to the desired local schema, as well as the ultimate propagation of the public data source update to the local view.

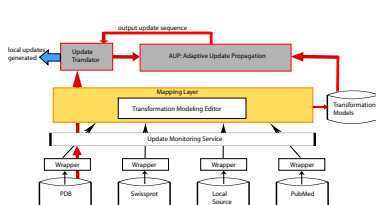


Fig. 1 An Architectural Overview of Sangam.

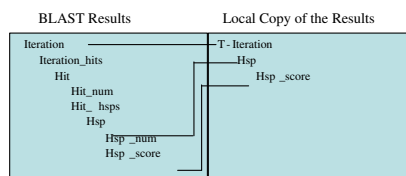


Fig. 2 Mapping of BLAST Result DTD Elements to the local MyData Schema using a Box and Line Method.

Step 1: At the core of the *Sangam* framework is the *mapping layer*. It supports the construction and management of explicit schema mappings, enabling scientists to define *local views* from one or more public data sources. Figure 2 depicts a fragment of the mapping between the BLAST [AGM⁺90] result schema and a locally defined schema using a basic box and line method. We should note here that results from BLAST can be (and are often) stored locally on the client machine in a variety of formats ranging from XML, FASTA, ASN.1, and plain text to in some

cases the relational format. These results maybe stored verbatim or a client (scientist) may choose to store only pertinent subsets of information elucidated from the results as shown in Figure 2. In *Sangam* the lines between the schema elements are explicitly represented and subsequently stored using the *transformations*. In general, the transformations can be augmented with XQuery [W3C01] fragments to provide translation semantics for the underlying data. However, we argue that in the life sciences domain, it is unreasonable to expect that the scientists would have a working knowledge of the intricacies of XQuery. Hence in this version of the system we provide only simple predicate selections for the mapped elements. Defining the explicit mappings between the local and public sources constitutes the first step of the process.

Step 2: The key focus of *Sangam* framework is to facilitate the propagation of pertinent updates from the *subscribed* public data source to the local view of the data stored at the user site. The first step of this process is an *alerter* service that monitors the public data sources and is able to detect any and all updates. Multiple approaches at detecting updates in XML files have been studied in the literature [CRGMW96, NACP01, XWW⁺02, CAM02, WDC03], that are mostly based on comparing consecutive database versions. While these approaches are general, they provide a good first step in this direction. And while we recognize the importance of this step, it is not the focus of this paper. Here we assume that this step is accomplished manually, and no further discussion of this is provided in this paper.

Step 3: Once relevant update data has been extracted from the public data source, the *update service* propagates this update from the public database to the local data view. This update is accomplished via the incremental propagation of the updates through the explicitly modeled transformations maintained by the mapping manager. In some cases, if the update is a simple data update with respect to the source database, then we employ a simplified version of the view maintenance algorithm for cross model update propagation. However, if the change entails schema structuring, then the transformation query itself may no longer be well-defined on the data source. The update service in this case not only propagates the update through the mapping, but also upgrades the connection by rewriting the transformation to once again become well defined—connecting the source to the target.

In the forthcoming sections, we detail the *mapping layer* and the *update service layer* and then show how the technology can be used to accomplish a real situation encountered by our collaborators in their work on studying the conservation of the α/β peripheral heme group.

3 Sangam Framework - The Transformation Layer

In this section, we briefly describe the two fundamental components of the *Sangam* middle layer, namely the middle layer schema model and the transformations that algebraically capture the restructuring/mappings between the schemas. Due to space constraints, we present these models based on examples, while full details can be found in [CR03b].

3.1 Schema Model

We assume, as in previous modeling approaches [AT96, GL98, PR95, BR00, MR83], that schemas from different data models are represented in one common data model. Our middle layer schema model, the *Sangam* model, is a graph-based model that is loosely based on the XML Schema model and can represent schemas from the XML, relational and object models³.

```
<!ELEMENT Iteration (
  Iteration_iter-num,
  Iteration_query-ID?,
  Iteration_query-def?,
  Iteration_query-len?,
  Iteration_hits? )>
<!ELEMENT Iteration_hits ( Hit* )>
<!ELEMENT Hit (
  Hit_num,
  Hit_hsp? )>
<!ELEMENT Hit_hsp ( Hsp* )>
<!ELEMENT Hsp (
  Hsp_num,
  Hsp_score )>
```

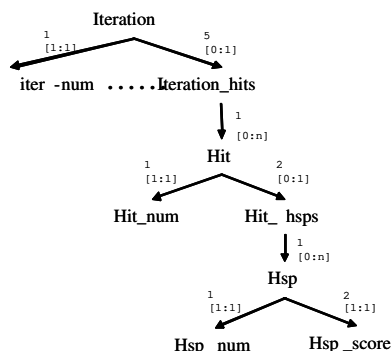


Fig. 4 Sangam Graph for the BLAST DTD given in Figure 3.

Fig. 3 A Fragment of the BLAST [AGM⁺90] Result DTD.

An instance of the Sangam graph model is called a *Sangam graph*. A Sangam graph $G = (N, E, \lambda)$ is a directed graph of nodes N and edges E , and a set of labels λ . Each node in the Sangam graph represents a *user-defined type* and is uniquely identified in the entire graph by its label l in λ . Two nodes are connected by a directed edge that represents a composition, a *HAS-A*, relationship between the nodes.

Example 1 Consider the BLAST-DTD fragment shown in Figure 3. Here the element *Iteration* has sub-elements *Iteration_iter-num* and *Iteration_hits* among others. The element *Iteration_hits* in turn is comprised of the subelement *Hit*. Figure 4 gives the Sangam graph representation for this BLAST-DTD fragment. The elements *Iteration*, *Iteration_iter-num*, *Iteration_hits*, and *Hit* are represented as nodes in the Sangam graph with the same labels. The *HAS-A* relationship between the element *Iteration* and its subelements *Iteration_iter-num* and *Iteration_hits* is represented by

³ Inheritance in object models is handled via flattening of the hierarchy.

directed edges from the node *Iteration* to the nodes *Iteration_iter-num* and *Iteration_hits* respectively. Similarly, an edge from the node *Iteration_hits* to *Hit* denotes the *HAS_A* relationship between the elements *Iteration_hits* and the *Hit*.

An edge between two nodes can be annotated with a set of constraints. The *local order* constraint specifies a monotonically increasing, relative local ordering for all outgoing edges from a given node *n*.

Example 2 Consider again the BLAST DTD shown in Figure 3. Here *Iteration_iter-num* and *Iteration_hits* are the first and the fifth subelements of the element *Iteration*. This is captured in the Sangam graph by the local order constraint resulting in an annotation of 1 on the edge between *Iteration* and *Iteration_iter-num*, and an annotation of 5 on the edge between *Iteration* and *Iteration_hits* as shown in Figure 4.

The *quantifier* constraint, $[m:n]$, on a given edge *e* between nodes *p* and *c* specifies the minimum *m* and maximum *n* occurrences of the data instances of the node *c* for a given instance of node *p*.

Example 3 In Figure 3, the subelements *Iteration_iter-num* and *Iteration_hits* are required and optional respectively, while the subelement *Hit* of the element *Iteration_hits* can have unbounded occurrences. This information is captured in the Sangam graph (Figure 4) by the quantifier constraint, and is denoted as $[1:1]$ for the required subelement (*Iteration_iter-num*), $[0:1]$ for the optional subelement (*Iteration_hits*) and $[0:n]$ for the unbounded subelement *Hit*.

A set of *Sangam data graphs* are associated with each Sangam graph to represent the extent of the graph. Each Sangam data graph conforms to the Sangam graph with which it is associated, and is analogous to a tuple in the relational model or a fragment of the XML document in the XML model. At each individual node and edge, the corresponding values in the Sangam data graphs are represented as *nodeObjects* and *edgeObjects* and are termed the extent of the node and the edge respectively.

Example 4 Figure 5 shows a Sangam data graph for the Sangam graph given in Figure 4. The corresponding fragment of the XML document is shown in Figure 6.

3.2 The Transformation Model

A key contribution of our work are the *transformations* that provide the foundation of the box and line restructuring shown in Figure 2. In this section, we now introduce the building blocks of this transformation language – the linear graph transformation operators and the glue logic that enables the composition of these operators to represent transformations over Sangam graphs.

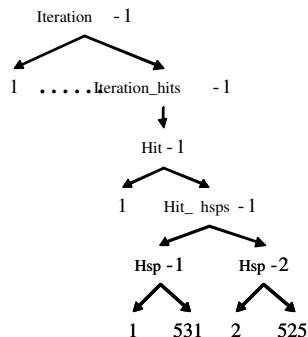


Fig. 5 A Data Graph Corresponding to the Sangam graph in Figure 4.

```

<Iteration>
  <Iteration_iter-num>1</Iteration_iter-num>
  <Iteration_hits>
    <Hit>
      <Hit_num>1</Hit_num>
      <Hit_hsp>
        <Hsp>
          <Hsp_num>1</Hsp_num>
          <Hsp_score>531</Hsp_score>
        </Hsp>
        <Hsp>
          <Hsp_num>2</Hsp_num>
          <Hsp_score>525</Hsp_score>
        </Hsp>
      </Hit_hsp>
    </Hit>
  </Iteration_hits>
</Iteration>
...
...
</Iteration>
  
```

Fig. 6 The XML Document Representation of the Data Graph in Figure 5.

Linear Transformation Operators. We define a set of linear transformation operators, namely the `cross` operator that represents a basic node-copy; a `connect` operator that represents an edge-copy; a `smooth` operator that reduces a path of two edges to a single edge; and a `subdivide` operator that expands a single edge to represent a path of two edges.

The `cross` operator, very simply stated, represents a node copy operator. It takes as input a node n and produces as output an exact copy n' . The `cross` operator is a total mapping, i.e., the nodeObjects in the extent of n are mapped one-to-one to the nodeObjects in the extent of n' .

Example 5 Figure 7 shows the application of the `cross` operator to the input node `Hsp_score`. The output of the `cross` operator is the node $T\text{-Hsp_score}$ ⁴. All nodeObjects, namely nodes in the data graphs that correspond to the values 531 and 525 in the extent of the input node `Hsp_score` are mapped one-to-one to nodeObjects in the extent of the output node $T\text{-Hsp_score}$. We use the notation \otimes with an incoming and outgoing edge to graphically represent the `cross` operator. The incoming edge denotes the input while the outgoing edge denotes the output produced by the node.

The `connect` operator represents an edge copy operator. It takes as input an edge e between two nodes m and n , and produces as output a copy e' between two

⁴ We use the prefix of T - to easily distinguish the target nodes and edges. We follow this naming convention throughout the paper.

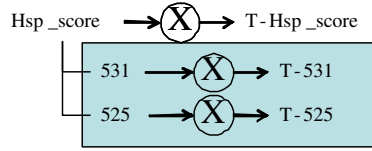


Fig. 7 An Example Showing the Application of Cross Linear Transformation Operator to an Input Node. The Corresponding Data Graph Transformation is also Shown.

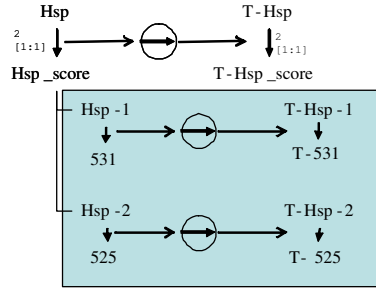


Fig. 8 An Example Showing the Application of a Connect Linear Transformation Operator to an Input Edge. The Corresponding Data Transformations are also Shown.

nodes m' and n' . The annotations of e' (the order and the quantifier constraints) are the same as the annotations of the input edge e . The connect operator is a total mapping, i.e., the edgeObjects in the extent of the edge e are mapped one-to-one to the edgeObjects in the extent of e' .

Example 6 Figure 8 shows the application of the connect operator to the input edge Hsp_score between the nodes Hsp and Hsp_score . The output of the connect operator is the edge $T-Hsp_score$ between nodes $T-Hsp$ and $T-Hsp_score$. All edgeObjects, namely the edges in the data graphs that correspond to the edges between the nodes $Hsp-1$ and 531 , and the nodes $Hsp-2$ and 525 are mapped one-to-one to the edgeObjects in the extent of the $T-Hsp_score$. We use the notation \oplus with an incoming and an outgoing edge to graphically represent the connect operator. The incoming and the outgoing edges here represent the input and the output of the operator.

The smooth operator is a path reduction operator. It reduces a given input path of length 2 to an output path of length 1 - that is it produces as output a single edge. It can, however, be applied repeatedly for the reduction of paths with larger lengths. The annotations of the output path (single edge) are modified from those of the input path (of length 2) to preserve the information capacity during the reduction. The local order annotation of the output path reflects the local order annotation of the first edge of the path. This stems from the fact that the two paths (input and output) logically originate from the same node. The quantifier annotation, however, in the output path is modified to represent the combined capacity of the input path. Intuitively, if both the edges in the input path have a $[1:1]$ quantifier constraint, then the output path would be similarly constrained. However, if the first edge of the input path is $[1:1]$, and the second edge $[1:m]$, then the quantifier annotation of the output path must reflect the combined capacity of the input path and be set to $[1:m]$ in this case. The smooth operator is applied in a similar manner to the paths in the data graphs to create paths of length 1 in the output data graphs.

Example 7 Figure 9 shows the application of the smooth operator to the input path between the nodes `Hit_hsp`, `Hsp` and `Hsp_score` to produce the output path of length 1 between the nodes `T-Hit_hsp` and `T-Hsp_score`. A similar reduction is applied to the corresponding paths in the data graphs. We use the notation \ominus with two incoming edges and one outgoing edge to represent the smooth operator. The two incoming edges denote the input path of length 2, while the outgoing edge denotes the reduced path.

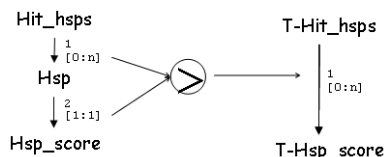


Fig. 9 An Example Showing the Application of the Smooth Linear Transformation Operator to Two Input Edges.

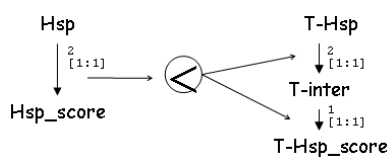


Fig. 10 An Example Showing the Application of a Subdivide Linear Transformation Operator to an Input Edge.

The subdivide operator intuitively performs the inverse of the smooth operator. It expands a given path of length 1 to produce a path of length 2 – that is, it splits a given edge between two nodes to produce two edges and an intermediate node as output. The local order annotation for the first edge in the output path reflects the local order annotation of the input path, as the edges logically stem from the same node. The local order annotation of the second edge in the output path is set to 1 to denote the only outgoing edge from the newly created intermediate node. The quantifier annotation for the output path is set to reflect the information capacity of the input path. Intuitively, the second edge of the output path now reflects the information capacity of the input path. Hence, its quantifier is set to be the same as that of the input path. The quantifier for the first edge must, however, reflect at the least the minimum capacity of the input path. Hence, the quantifier for the first edge of the output path is set to $[0 : 1]$ if `min` for the input path is 0, else it is set to $[1 : 1]$. A similar subdivide operation is applied on all corresponding edges in the data graphs.

Example 8 Figure 10 shows the application of the subdivide operator. Here the subdivide operator expands the path (of length 1) between the nodes `Hsp` and `Hit_score` to produce as output a path of length 2 between the node `T-Hsp`, the newly created intermediate node `T-inter` and the node `T-Hit_score`. A similar operation is applied on the corresponding edgeObjects in the data graphs, and intermediate objects are created as needed. This corresponds to the splitting of the edgeObject into two separate edgeObjects and the creation of a new nodeObject.

Composing the Linear Transformation Operators. The linear transformation operators are primitives that operate on the nodes and edges of a given graph. These operators while complete in the transformations they produce [GY98], are

not sufficient to express complex transformations that can effectively reduce any given input graph to a desired output graph. Glue logic that binds these operators together is essential to accomplish this. We now introduce two such bindings, namely a context dependency and a derivation binding.

Context dependency is a correctness binding that enables several linear transformation operators to collaborate and jointly operate on sub-graphs of the input graph to produce one combined output graph. The output of all linear transformation operators bound by context dependency is visible in the output graph. We term the context dependency as a correctness binding as its definition is used to produce the correct set of data graphs.

Example 9 Figure 11 shows an example of a context dependency binding. Here, the sub-graph G is mapped using cross and smooth operators bound by context dependency to produce the output graph G' . Here, one cross operator produces the output node $T\text{-Hit_hsp}$ s based on the input node Hit_hsp s; the second cross operator produces the output node $T\text{-Hit_score}$ based on the input node Hit_score ; and the smooth operator produces an output edge based on the input path between the nodes Hit_hsp s, Hsp and Hit_score . The context dependency binding ensures that the output edge connects the nodes $T\text{-Hit_hsp}$ s and $T\text{-Hit_score}$ that are the mappings of the nodes Hit_hsp s and Hit_score produced by the respective cross operators. Similarly, the context dependency ensures that (1) the corresponding data graphs reflect the binding of the node $T\text{-Hit_Hsp}$ s-1 to node $T\text{-531}$, and the node $T\text{-Hit_Hsp}$ s-2 to node $T\text{-525}$ respectively; and (2) the node $T\text{-Hit_Hsp}$ s-1 is mapping of Hit_Hsp s-1, node $T\text{-Hit_Hsp}$ s-2 is mapping of Hit_Hsp s, and nodes $T\text{-531}$ and $T\text{-525}$ are mappings of the nodes 531 and 525 respectively produced by the cross operators.

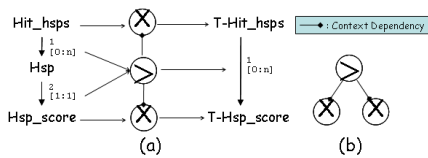


Fig. 11 Building a Transformation Using a Context Dependency Binding. Here (a) shows the actual transformation with the transformation operators and their respective inputs and outputs, while (b) shows a thumbnail of the transformation.

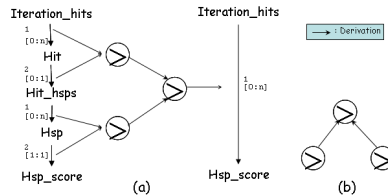


Fig. 12 Building a Transformation Using Derivation Bindings. Here (a) shows the actual transformation with the transformation operators and their inputs and outputs, while (b) shows a thumbnail of the transformation.

Derivation binding enables the sequential application of several linear transformation operators via operator nesting. In this binding, the output of one operator is pipelined into the next operator, and the final output is the output of the parent operator that consumes the inputs of its children operators.

Example 10 Figure 12 shows an example of a derivation binding. Here, a path of length 4 is mapped to a path of length 1 using three smooth operators. The first two smooth operators take as input paths of length 2 between the nodes *Iteration_iter-num* and *Hit* and *Hit_hsp*s and *Hit_score* respectively. These operators produce two paths of length 1 that are then pipelined into the third smooth operator to produce a path of length 1 between the nodes *T-Iteration_iter-num* and *Hit_score*. The third smooth operator here consumes the output of the first two smooth operators and produces the final output.

Both context dependency and derivation bindings together with the transformation operators result in *tree-like* structures providing a form analogous to algebra trees for the transformation language.

Summary. Both context dependency and derivation bindings can and in many cases must co-exist to provide a correct transformation of both the Sangam graphs and the corresponding data graphs. Figure 13 depicts the transformation corresponding to the mapping example given in Figure 2. Here both context dependency and derivation bindings are utilized to produce the final desired output. It should be noted here that the transformation language presented here is simple yet powerful enough to express all linear transformations on first-order graphs. These transformations are independent of the data model of the actual source and target. For example, the transformation in Figure 13 can be applied to both XML and FASTA format results obtained from BLAST, and the resulting output can be stored as either an XML document, a FASTA file or as a relational database. Moreover, the transformations can be translated to high-level languages, such as XQuery or SQL, to perform the actual data transformations.

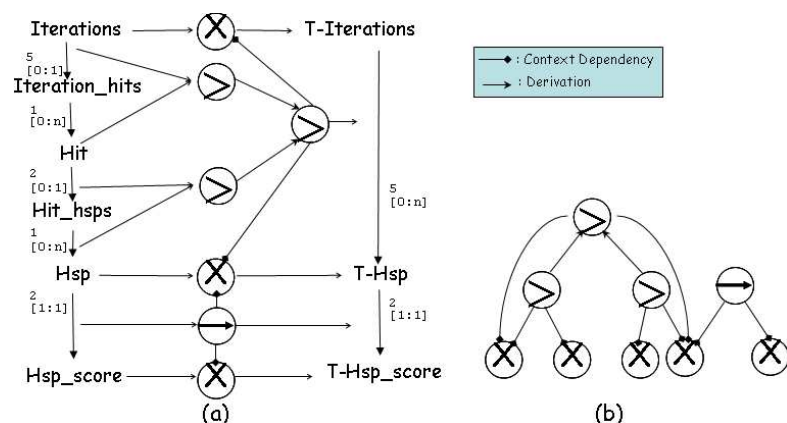


Fig. 13 Transformation Corresponding to the Box-Line Transformation Shown in Figure 2. Here (a) shows the transformation with all inputs and outputs, while (b) is the thumbnail sketch of the transformation.

Schema Change Primitive	Description
<code>insertNode (ln, τ, m, l, q)</code>	Creates new node with label <code>ln</code> and inserts it as child of node <code>m</code>
<code>insertNodeAt (ln, τ, m, l, q, pos)</code>	Creates new node with label <code>ln</code> and inserts it as child of node <code>m</code> at position <code>pos</code>
<code>deleteNode (m, n)</code>	Deletes child node <code>n</code> from parent <code>m</code>
<code>insertEdge (m, n, l, q)</code>	Inserts new edge between nodes <code>m</code> and <code>n</code> , making <code>n</code> child of node <code>m</code>
<code>insertEdgeAt (m, n, l, q, pos)</code>	Inserts new edge at position <code>pos</code> between nodes <code>m</code> and <code>n</code> , making <code>n</code> child of node <code>m</code>
<code>deleteEdge (m, l)</code>	Deletes edge <code>e</code> with label <code>l</code> from node <code>m</code>
<code>rename (n, l')</code>	Modifies label of node <code>n</code> to <code>l'</code>

Table 1 Taxonomy of Sangam Graph Change Primitives.

4 Taxonomy of Sangam Graph Changes

Information is not static, rather it changes over time. In most database systems this change is made through a set of data update primitives and schema evolution primitives, the semantics of which define the precise change. We take the same approach for enabling changes on the Sangam graphs. We introduce five Sangam graph change primitives, namely `insertNode`, `deleteNode`, `insertEdge`, `deleteEdge` and `rename`, to insert and delete a node, insert and delete an edge, and to rename both the node as well as an edge respectively. Table 1 gives a brief description of these operations.

We define a similar set of four operations to enable modifications on the data graphs. These primitives capture the semantics for the correct modification of data graphs via the insertion and deletion of `nodeObjects` and the `edgeObjects`. Table 2 gives a brief description of these Sangam data graph modification operations. It can be shown that the Sangam graph change primitives as well as the Sangam data graph change primitives are both minimal and correct [Cla02].

5 The Adaptive Update Propagation Strategy

Once a local change has been translated into a set of changes on the Sangam graph in the middle layer, it must then be translated into a set of changes on the target Sangam graph. This is in essence similar to the incremental view maintenance strategies proposed in literature [BLT86, GM95, MKK97, ZGMHW95, AESY97], albeit with some key differences. For SQL views, the focus is on the calculation of the extent difference between the old V and new view V' by adding or subtracting tuples from the extent of the view V . In these works, the schemas of V and V' are assumed to stay the same. In the scenario when schema changes are considered [KR02a] as in update propagation through SchemaSQL views by Koeller et al., the changes are made in-place on the individual algebra operators in the algebra

Data Modification Primitive	Description
addObject (v, n)	Creates new object o with data value v . Inserts object o into extent $I(n)$ of node n
deleteObject (o, n)	Removes object o from extent $I(n)$ of node n based on either the object id or the value
addEdgeObject (o_1, o_2, e)	Creates new edgeObject ($o_e: \langle o_1, o_2 \rangle, ord$). Inserts the edgeObject o_e into the extent $R(e)$ of edge $e: \langle m, n \rangle$ at position ord
deleteEdgeObject (o_e, e)	Removes object o_e from extent $R(e)$ of edge $e: \langle m, n \rangle$ based on either the object id or the value

Table 2 Taxonomy of Sangam Data Graph Change Primitives.

tree. That is, the structure of the algebra tree remains the same and no algebra nodes are added or deleted.

However, when propagating Sangam graph changes through a transformation not only is the old output Sangam graph G different from the modified output Sangam graph G' , but in many cases the structure of the transformation itself may be affected. For instance, new transformation operators may be added as a result of a node insertion and/or existing transformation operators may be removed as a result of a node or an edge deletion in the Sangam graph. To address this, we introduce the **Adaptive Update Propagation (AUP)** algorithm – a two pass algorithm. The first pass of **AUP** propagates the update while the second pass performs any needed maintenance on the transformation itself.

The First Pass: Propagating the Update. The first pass of the **AUP** algorithm propagates the given update, in the form of a change primitive, through the transformation. The algorithm performs a post-order traversal to ensure that each transformation operator processes the updates after all its children have already computed their output. The result of the first pass of the **AUP** algorithm is the update sequence δ , that is applied to the output Sangam graph at the end of the update propagation.

Figure 14 depicts the first pass of the **AUP** algorithm. Here, a single update u is applied to each transformation operator. Each transformation operator can determine if its output is effected by the update. If its output is affected, it produces an update, u_i' that must be applied to its output node or edge in the output Sangam graph. The change together with the transformation operator are recorded as a pair $\langle u_i', op_i \rangle$ in the *update sequence* δ produced by the transformation operator. Both the original update u_i and the generated update sequence δ are then propagated through the transformation along the context dependency and derivation bindings.

While propagation is carried out through both context dependency and derivation bindings, and applied to the parent operator, the type of binding decides whether: (1) the original update u must be applied to the parent operator; and (2) whether the update sequence generated by the child operator must be propa-

```

func List AUP(Update  $u_p$ , Operator  $op_p$ ) {
  List  $\delta$   $\leftarrow \emptyset$ , //Updates from children
  List  $\delta' \leftarrow \emptyset$  // parent update sequence
  UpdatePair  $up$ ,  $up_i$ 
  Boolean updateApplied
  if ( $op_p$ .isLeaf)
     $up \leftarrow \text{applyUpdate}(u_p, op_p)$ 
     $\delta.append(up)$ 
  else
    //If not a leaf, then recursively invoke
    // AUP for all children
    for (all children  $op_i$  of  $op_p$ )
       $\delta.append(\text{AUP}(u_p, op_i))$ 
      //  $\delta =$  updates of children of  $op_p$ 
      // Calc. effect of each update on  $op$ 
      // Generate parent's  $\delta'$ 
      for (all updatePairs  $up_i \in \delta$ ) {
         $\delta' \leftarrow \text{calculateUpdate}(up_i, op)$ 
      }
    return  $\delta'$ ;
}

func List calculateUpdate(UpdatePair  $up_i$ ,
  Operator  $op$ ){
  List  $\delta' \leftarrow \emptyset$ 
   $u_i \leftarrow up_i.getUpdate()$ 
   $up_i' \leftarrow \text{applyUpdate}(u_i, op_p)$ 
  //Decide if  $up_i$  and  $up_i'$  appended
  //to parent's  $\delta'$  or  $up_i$  discarded
   $op_i \leftarrow up_i.getOperator()$ 
  if ( $(e:\langle op, op_i \rangle) ==$ 
    contextDependency)
    // Calculate effect of original update
    // Check if update applied before
    if (!updateApplied)
       $up \leftarrow \text{applyUpdate}(u_p, op_p)$ 
      updateApplied  $\leftarrow$  true
       $\delta'.append(up_i)$ 
       $\delta'.append(up)$ 
       $\delta'.append(up_i')$ 
    elseif ( $(e:\langle op, op_i \rangle) ==$  derivation)
       $\delta'.append(up_i')$ 
  return  $\delta'$ 
}

```

Fig. 14 First Pass of Cross Algebra Tree Target Maintenance Algorithm.

```

func UpdatePair applyUpdate(Update  $u_p$ , Operator  $op_p$ )
{
  if ( $op_p.isAffectedBy(u_p)$ )
     $u_p' \leftarrow op_p.generateUpdate(u_p)$ 
    if ( $op_p.toBeDeleted()$ )
       $op_p.markForDeletion()$ 
     $up \leftarrow (u_p', op_p)$ 
  return  $up$ 
}

```

Fig. 15 First Pass of Cross Algebra Tree Target Maintenance Algorithm - The UpdatePair Function.

gated up, i.e., whether it should be included in the update sequence generated by the parent operator. For a context dependency binding, recall that output of all operators is visible in the output graph. Hence, if the propagation is via a context dependency binding the original update u is applied to the parent operator and any update produced by the parent operator is *appended* to the update sequence δ produced by the child operator. In the case of derivation binding, recall that any intermediate results produced by the children operators are not visible in the output graph. Moreover, the parent operator's output is dependent on the input provided by its children operators. Hence, when updates are propagated through the derivation binding only updates produced by the children operators are applied to the parent operator. The original update is not applied to the parent in this case. Fur-

thermore, the update sequence δ produced by the children operators is discarded and a new update sequence δ' containing only the parent's update is produced.

For any transformation operator op_i if the update u_i is such that it removes its input, then the operator op_i is marked for deletion. The actual removal of the operator from the transformation occurs in the second pass, *AUP_Clean*, of the *AUP* algorithm.

The Second Pass: Cleaning Up. The second pass of the *AUP* algorithm removes any non-functional operators from the transformation, and if needed modifies the update sequence δ' produced by the first pass of the algorithm to reflect the effects of the removal. Figure 16 gives the second pass of the *AUP* algorithm – *AUP_Clean*. There are two cases that must be considered here: (1) an operator with context dependency bindings has been marked for deletion. In this case, as the outputs of all other operators are independent from any other output, no other operator is effected; and (2) an operator with derivation bindings to its children operators has been marked for deletion. In this case, as only the output of the root operator is visible in the output graph, the derivation binding along with all involved transformation operators are deleted⁵. Any effects of the deletion on the update sequence δ' are computed and the final update sequence δ'' is applied to the output graph.

```

func AUP_Clean (Operator op)
  if (op.markForDeletion () )
    // Propagate markForDeletion to all
    // derivation children
    for (all children  $op_i$  of op)
      if ((e:<op,  $op_i$ >) == derivation)
        if ( $op_i$  is not shared)
          // Mark the child operator for deletion.
           $op_i$ .markForDeletion()
        else
          // Mark the derivation edge for deletion.
          e.markForDeletion()
    //Recursively invoke algorithm
    for (all children  $op_i$  of op)
      AUP_Clean ( $op_i$ , op)
    // Remove the operator
    if (op.markForDeletion () )
      delete (op)
  return

```

Fig. 16 Second Pass - Clean Up of Cross Algebra Tree Target Maintenance Algorithm.

The *AUP* algorithm has been shown to be correct, that is (1) the propagation of an update u by the *AUP* algorithm results in a well-formed transformation – this shows the correctness of the second phase of the algorithm; (2) the propagation of an update u by the *AUP* algorithm results in a valid modified output Sangam graph; and (3) the incremental update propagation algorithm *AUP* produces an

⁵ It should be noted here that if a child operator in either bindings is marked for deletion, it would produce an update deleting its output in the first phase – propagating the deletion to the parent nodes as needed. The impact of its deletion is therefore covered in the first phase of *AUP*.

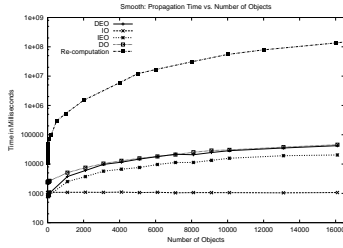


Fig. 17 Time Taken to Incrementally Propagate the `insertObject(IO)`, `deleteObject(DO)`, `insertEdgeObject(IEO)` and `deleteEdgeObject(DEO)` operations, Through One Smooth Algebra Operator Compared to the Cost of Re-computation. The number of objects here refers to the size of the extent of the given Sangam graph.

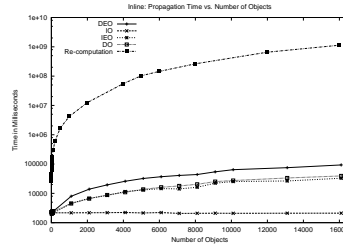


Fig. 18 Time Taken to Incrementally Propagate the `insertObject(IO)`, `deleteObject(DO)`, `insertEdgeObject(IEO)` and `deleteEdgeObject(DEO)` Operations Through Complex Transformation of over 10 Operators Compared to the Cost of Re-computation. The number of objects here refers to the size of the extent of the Sangam graph.

updated Sangam graph equivalent to that produced by full re-computation. Full proof of this is beyond the scope of this paper. We refer the interested reader to [Cla02].

Experimental Evaluation of AUP We conducted several experiments to evaluate the performance of the *AUP* algorithm. The experiments focused on two primary aspects: (1) the relative cost of propagating the different types of updates through a given transformation. Figure 17 shows the time taken to incrementally propagate different data update operations through the `smooth` operator; and (2) the cost of incremental propagation (the *AUP*) algorithm compared to the cost of full re-computation. In both cases only one update was propagated through the transformation at a time. Figure 18 shows the cost of full re-computation in comparison to the incremental propagation of different updates. In both Figures 17 and 18, the extent of the Sangam graphs was increased. The *AUP* algorithm exhibits a logarithmic increase with a linear increase in the transformation size⁶ for the propagation of updates such as `DeleteEdgeObject` and `InsertEdgeObject`. The algorithm is nearly constant with linear increase in transformation size for `DeleteObject` and `InsertObject` primitives. In all cases re-computation was found to be more expensive.

6 Case-Study: From Mapping to Propagation

In this section, we present a case-study explicating our overall technique to accomplish cross-data model propagation of updates from a public data source to a

⁶ The transformation size here refers to the complexity of the transformation measured in terms of the number of operators.

local view. For this case-study we use an example based on the ongoing work of Dr. McDonald and Dr. Vasudevan of the Chemistry Department at UMass-Lowell. Their research focuses on statistical analysis of amino acids essential to heme incorporation. This is a molecular phylogenetic approach to the apparent conservation of heme contact residues in both primary sequence and, presumably, tertiary (heme pocket) topology. Current on-line databases and retrieval tools are used to evaluate the sequence conservation of the α/β peripheral heme group contacts as a first step.

6.1 The Mapping Layer

To facilitate this investigation, we built a local data set of sequences from the α and β chains of only those organisms that assemble a tetrameric ($\alpha_2\beta_2$) protein. The local schema, mapped from the XML Schemas of Swissprot [BA00] (SPTR.xsd found at [SPT04]), and PDB (PDBj.xsd found at [PDB04]), defined the key information that was of importance to our collaborators. Here the PDB data was used to discover the sequence IDs of the α and the β sequences, while Swissprot was the main source for the sequences themselves. An additional restriction limiting the sequences to a length of 141 amino acid residues was imposed on the corresponding data transformation. Figure 19 represents the box and line mapping of the relevant Swissprot and PDBj elements to the local schema. The boxes capture the predicate selection that is applied on, for example, the sequence length. This is the view of the *Sangam* framework that will typically be used by the life scientists.

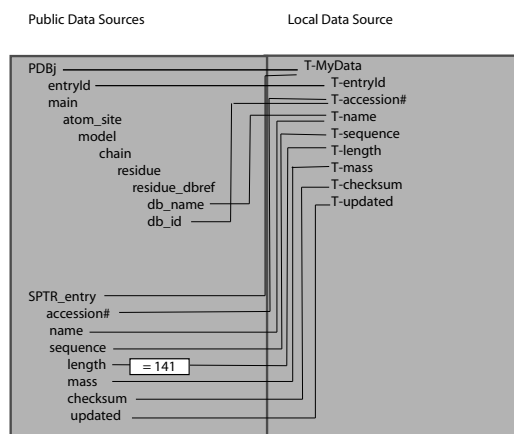


Fig. 19 Mapping of Swissprot and PDBj Elements to the local T-MyData Schema Shown Using the Box and Line Method.

Figures 20 and 21 depict the graph based representation of the Swissprot and PDBj fragments shown in Figure 19, while Figure 22 depicts the same for the local

schema fragment in Figure 19. We use the prefix of T- in Figures 19 and 22 to uniquely identify the target elements in our discussion here.

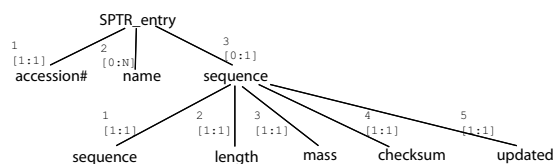


Fig. 20 Sangam's Graph-based Representation of the Swissprot Fragment shown in Figure 19.

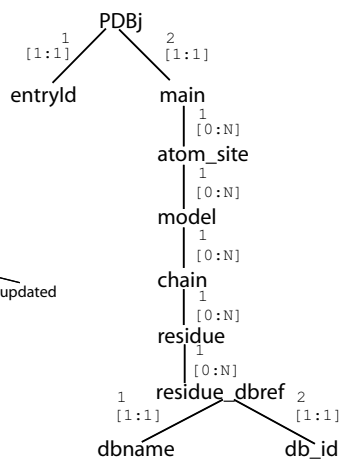


Fig. 21 Sangam's Graph Based Representation of the PDBj Schema Fragment shown in Figure 19.

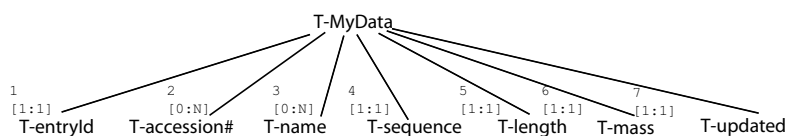


Fig. 22 Sangam's Graph Based Representation of the local T-MyData Schema.

Figure 23 represents the transformation corresponding to the mapping shown in Figure 19. Here, both context dependency and derivation bindings are utilized to produce the final desired output. The desired input nodes (such as entryId and db_name) are copied, using the *cross* transformation operator, to produce the desired output nodes.

The edges between the newly created output nodes are established via either direct copying (*connect* operator) of the correct edges from the input graph, or via the *smooth* transformation operator. For example, the edge between the parent PDBj and the child entryId in the source pane is represented identically by the edge between the elements T-MyData and T-entryId. This is an edge copy that can be accomplished by a *connect* operator. To ensure that the edge is established between the right nodes and correct data graphs are generated, the cross operators for PDBj:T-MyData and entryId:T-entryId are placed in a context dependency binding with the *connect* operator as shown in Figure 23(a).

On the other hand, we must reduce a path of length 7 between the `PDBj` and the `db_id` to establish a path of length 1 between the nodes `T-MyData` and `T-accession#`. This transformation is accomplished by six smooth operators that successively reduce the input path to the desired output path. The smooth operators are glued together with a derivation binding. In addition, context dependency bindings between the root smooth operator and the cross operators (`PDBj:T-MyData`, and `db_id:T-accession#`) are used to establish the path between the right nodes. Figure 23(b) depicts the resulting transformation. The transformation for the Swissprot Schema is established in a similar manner and is given in Figure 23(c).

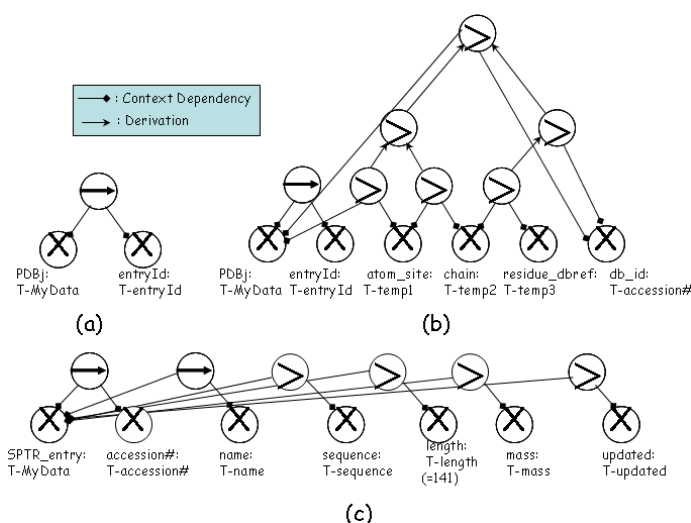


Fig. 23 The Transformation Representing the Mapping Presented in Figure 19.

The predicates depicted in Figure 19 are captured as annotations on the corresponding mapping of the target (local data view) element(s). For example, the restriction that sequence length be 141 residues is captured as an annotation on the cross operator that maps the `length` in the source pane to the `T-length` in the target pane. Such restrictions must be explicitly defined by the user. However, some restrictions can be inferred from the mapping provided by the user – these are the implicit constraints. For example, the mapping of both source elements `db_id` and `accession#` to the target element `T-accession#` places an implicit join constraint on the mapping that is enforced when transforming the data graphs.

We use an XML-based storage model for persisting the schema models as well as the transformation models. The local data set in this case-study was stored as a relational database using the MySQL DBMS.

6.2 Propagating Updates

6.2.1 Data Updates During the course of this case study (Jan 2004 to May 2004), we observed only **data updates** in Swissprot and PDB that conformed to the mapping shown in Figure 19. At the beginning of the case-study (Jan 2004), the mapping resulted in 152 unique sequences for the α chain, while at the end of the case study (May 2004) there were a total of 177 sequences for the α chain in the local database. Figures 24 and 26 show example updates (in XML format) that were received from the source databases (Swissprot and PDB). The corresponding update data graphs, the virtual representation in the middle-layer, for these example updates are shown in Figures 25 and 27 respectively. Figures 28 and 29 depict partial update sequences, $\delta - 1$ and $\delta - 2$, generated from the the data graphs in Figure 25 and 27 respectively.

```
<entry dataset='Swiss-Prot'
  created='1986-07-21'
  updated='2004-07-05'>
  <accession>P01999</accession>
  <name>HBA_ALLMI</name>
  ....
  .....
  <sequence length='141'
    mass='15734'
    checksum='F4A95E87BB7ACD86'
    updated='1986-07-21'>
    VLSMEDKSNVKAIWGKASGHLEEYGAEL
    ERMFCAYPQTKIYFPHFDMSHNSAQIRAH
    GKKVFSALHEAVNHIDDLPGALCRLSEL
    HAHSLRVDPVNFKFLAHCVLVVFAlHHPS
    ALSPEIHASLDKFLCAVSAVLTSKYR
  </sequence>
</entry>
</uniprot>
```



Fig. 25 The Corresponding Data Graph - Representation of the Update in Sangam.

Fig. 24 Update From Swissprot [BA00] Database.

Propagating the Updates. To exemplify our propagation algorithm, we now show how one of these updates — `addObject("141", length)` — is propagated through the fragment of transformation shown in Figure 30. The *AUP* algorithm follows a post-order traversal of the transformation implying that the update $u_1 = \text{addObject}("141", \text{length})$ is applied first to the leaf level cross operators, and then to the smooth operator.

Two factors determine the applicability of the update u_1 to any operator: (1) the match between the specified node in u_1 and the input of the operator; and (2) the specified predicate clause, if any. The application of the update u_1 to cross operator `SPTR_entry:T-MyData` does not satisfy either of these two factors, and hence results in an empty update sequence. The update u_1 is, however, applicable for the operator `length:T-length`. Here, there is a match between the operator's input node `length` and the node specified in the update u_1 . Moreover,

```

<entry dataset='PDBj'
  created='1986-07-21'
  updated='2004-07-05'>
  <PDBJ entryId='1A00'>
  <chain> A </chain>
  <residue>
    <residue_dbref> SWS </residue_dbref>
    <db_name> HBA_ALLMI </db_name>
    <db_id> P01999 </db_id>
  </residue>
  </chain>
</PDBJ>

```

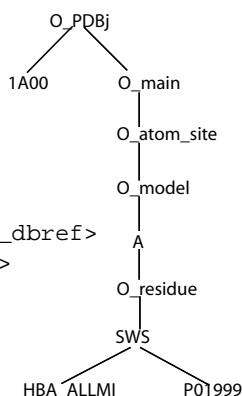


Fig. 26 Update From the PDBj Database.

Fig. 27 The Data Graph Corresponding to the PDBj Update.

Update 1: $o_1 = \text{addObject}(\text{"P01999"}, \text{accession\#})$
 Update 2: $\text{addEdgeObject}(o_1, o_{SPTR_entry}, \langle SPTR_entry, \text{accession\#} \rangle)$

Update 5: $o_2 = \text{addObject}(\text{"HBA-ALLMI"}, \text{name})$
 Update 6: $\text{addEdgeObject}(o_2, o_{SPTR_entry}, \langle SPTR_entry, \text{name} \rangle)$

Update 7: $o_3 = \text{addObject}(\text{"VLS-KYR"}, \text{sequence})$
 Update 8: $\text{addEdgeObject}(o_3, o_{SPTR_entry}, \langle SPTR_entry, \text{sequence} \rangle)$

Update 9: $o_4 = \text{addObject}(\text{"141"}, \text{length})$
 Update 10: $\text{addEdgeObject}(o_4, o_3, \langle \text{sequence}, \text{length} \rangle)$

Fig. 28 The Partial Update Sequence $\delta - 1$ Produced by the Updates in Figure 24 and Corresponding to the Update Data Graph in Figure 25.

Update 3: $o_x = \text{addObject}(\text{"P01999"}, \text{db_id})$
 Update 4: $\text{addEdgeObject}(o_x, o_{SWS}, \langle \text{residue_dbref}, \text{db_id} \rangle)$

Fig. 29 The Partial Update Sequence $\delta - 2$ Produced by the Updates in Figure 26 and Corresponding to the Update Data Graph in Figure 27.

the predicate selection restricting the sequence length to 141 residues also holds. The application of the update u_1 to this cross operator thus produces the update sequence $\delta_2 = \{ \langle \text{addObject}(\text{"141"}, \text{T-length}), o_{p_2} \rangle \}$.

As a next step, the union of the update sequences of the children, in this case simply δ_2 , together with the original update u_1 are applied to the parent operator (the smooth operator in Figure 30). Neither the update in δ_2 , nor the original update u_1 satisfy the two factors with respect to the smooth operator. However, the smooth operator is in a context-dependency relation with the cross operator

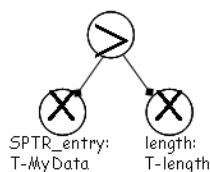


Fig. 30 The Transformation for length of a sequence.

length:T-length, the producer of an update, and hence the final update sequence $\delta = \delta_2$.

As there is no modification to the transformation structure, the second pass of the *AUP* algorithm is not invoked and the final update sequence δ is applied to the output model shown in Figure 22.

Updates with Explicit Constraints. The current taxonomy of Sangam Graph data changes (Table 2), does not include a *modification primitive*, as most data updates can be achieved by a combination of `delete` and `add` operations. Consider a modification of the sequence length (T-length) from 141 to 140 applied to the data graph given in Figure 31. There are two possible update outcomes in this case: (1) the operation is disallowed as the deletion of the sequence length violates the explicit constraint – the sequence length must be 141. This, however, does not accurately represent the updates in the source. The Sangam system is currently limited to supporting these semantics; (2) the operation results in the deletion of the entire corresponding data graph to both reflect the change in the source and maintain the semantics of the transformation. These semantics require the introduction of a new *modification primitive* that handles the deletion and addition as an atomic operation and deletes the entire data graph to ensure constraint conformance by all data graphs. Based on discussions with our collaborators, it is, however, unclear if automatic updates that result in the deletion of complete data graphs are the preferred semantics in all cases.

6.2.2 Schema Updates While during the case-study we did not encounter any real schema/structural updates, we now use an example to show how a schema update would be propagated through the transformation shown in Figure 23(a). Consider a schema update $u_3 = \text{deleteNode}(\text{PDBj}, \text{entryId})$ that deletes the child node `entryId` of the parent node `PDBj`. The propagation of this update through the transformation shown in Figure 23(a) begins at the cross operator that maps `PDBj` to `T-MyData`. This update has no effect on this operator and hence no update sequence is produced. The update when applied to the cross operator that maps `entryId` to `T-entryId`, however, is effected as its input has been deleted. This cross operator thus produces the update sequence $\delta = \text{deleteNode}(\text{T-MyData}, \text{T-entryId})$ to delete the corresponding node `T-entryId` in the output graph. As this operator no longer produces an output, it is marked for deletion.

As the two cross operators are bound to the connect operator via context dependency, both the original update as well as the update sequence δ are applied to it. There is an implicit deletion of the edge between `PDBj` and `entryId` that is caused by the deletion of the node `entryId`, making the input of the connect operator invalid. It thus produces an explicit update `deleteEdge (T-MyData, e)`, where `e` is the label of the edge between `T-MyData` and `T-entryId`. This update is appended to the update sequence δ . Furthermore as the connect operator no longer produces an output, it too is marked for deletion.

The second phase of the algorithm is invoked at the end of the first phase, as there now exist two operators that have been marked for deletion. As there are no derivation bindings, no other operators are marked for deletion, and the cross and connect operators are both deleted. At the end of this phase the transformation is reduced to just the cross operator that maps `PDBj` to `T-MyData`. No other updates are generated during this phase and the final update sequence δ is applied to the output graph.

Schema Changes - Insert Operations. Schema updates that correspond to deletions and modifications of the input Sangam graph can be propagated through the *AUP* algorithm presented in this paper. However, we find that *schema inserts* into the input Sangam graph cannot be propagated through this algorithm. This is primarily due to the fact that any additional node or edge created in the input Sangam graph can be mapped to the output Sangam graph in many different ways or in some cases not at all. Typically the mapping of the new node or edge in the input Sangam graph must be done in accordance with the existing transformation. It is feasible to provide some default propagation rules and hence default mappings for handling these inserts. We provide some heuristics to accomplish this in [Cla02].

6.3 Translating the Updates

The application of the entire update sequence given in Figures 28 and 29 will result in a data graph given in Figure 31. This data graph can be converted via the application of simple rules to either a tuple(s) that can be inserted into an existing relational table(s), or into an XML document(s). As an example, Figure 32 depicts an XML document that can be constructed from the data graph in Figure 31. The corresponding schema graph (Figure 22) provides the tag information while the data graph itself provides the element values. These, XML document and tuple, can be merged with existing XML documents and or inserted into relational tables respectively using standard techniques for the same. It should be noted that for updates that result in the deletion of a complete data graph the propagated update is used to generate the corresponding relational or XML update.

7 Related Work

Schema Transformation and Modeling. Schema transformation and integration is an active research area [HMN⁺99, MZ98, FK99, MIR93, CJR98, RR87]. Rosenthal



Fig. 31 The Output Data Graph Generated From the Updates given in Figure 25 and 27.

```

<entry dataset='MyData'
  created='2004-08-21'
  updated='2004-10-01'>
  <T-MyData
    T-entryId='1A00'
    T-length='141'
    T-mass='15734'
    T-checksum='F4A95E87BB7ACD86'
    T-updated='1986-07-21'>
  <T-accession>P01999</T-accession>
  <T-name>HBA_ALLMI</T-name>
  <T-sequence> VLSMEDKSNVKAIWGKA
  SGHLEEYGAEAL
  ERMFCAYPQTKIYFPFDMSHNSAQIRAH
  GKQVFSALHEAVNHIDDLPGALCRLSEL
  HAHSLRVDVFNFKFLAHCVLVVFALHHP
  ALSPEIHASLDKFLCAVSAVLTSKYR
  </T-sequence>
  </T-MyData>
  </entry>
  
```

Fig. 32 XML Document Generated from the Update Graph given in Figure 31.

et al. [RR87] have proposed a fixed set of linear schema transformations for an extended ER model. A recent IBM project Clio [HMN⁺99] develops a partially automated tool for discovering mappings between two schemas. Clio focuses on generating SQL queries that perform both data transformations and the translation of target queries into source queries. Milo et al. [MZ98] approach the problem of data translations based on schema-matching. A number of translation algorithms have also been proposed, for example between XML-DTD and relational schemas [FK99].

Meta-modeling has been looked at as a middle-ware medium to model schema integration and data transformation [MR83, AT96, BR00, PR95]. Papazoglou et al. [PR95] propose a middle-layer meta-model to accomplish transformations between the OO and relational data models, using a set of pre-defined translation rules that can convert source data models to and from the middle-layer meta-model. Using translations as basic building blocks, they aim to automatically generate mappings from one given model to another at run-time. Atzeni et al. [AT96] have presented a framework to describe data models and application schemas. They focus on discovering translations between data models and hence application schemas. Bernstein et al. [BR00, MRB03] have proposed a meta-modeling framework to represent schemas in a common data model to facilitate many services including schema translations.

Change Detection and Subscriptions. In recent years, driven by the success of the Internet, semistructured data models and in particular XML have become increasingly popular in many domains including scientific domains such as protein chemistry. In relational databases, triggers in the source databases can be used to alert an integration system of updates [HN99]. Web-based databases are often uncooperative with both an integration effort and subsequent update alerts. Therefore, there

is great interest in development of algorithms that detect and recognize database changes, for example by comparing database versions [XWW⁺02, CRGMW96, CAM02, WDC03].

Other areas of interest are models and languages to represent changing semi-structured data, often with an emphasis on XML [PGMW95, CAW98, CTZ01, WZ03, MACM01]. Chawathe et al. [CAW98] also present a notion of subscribing to a data source. Some projects, such as the Xyleme project [MACM01, NACP01] focus on integrating a vast number of XML source databases.

Updating XML. Approaches to the problem of updating XML views are relatively recent and few. An initial paper by Tatarinov et al. [TIHW01] proposes an extension of the popular XQuery query language for XML [W3C01] by update primitives, such as Delete and Rename. It furthermore discusses the translation of XML updates (insertions and deletions) to SQL updates in a system in which XML documents are stored in relational databases. Braganholo et al. [BDH03] study the updatability of a subclass of XML views using a nested relational algebra. Their algorithm relies on rather strong assumptions such as that the algebraic representation of a view does not include any *Unnest* operators, and that *Nest* only occurs as the last operator.

Incremental View Maintenance. Incremental view maintenance deals with the problem of propagating source database updates to the derived data. Due to its importance for many applications, view maintenance in relational databases has been a heavily researched area [BLT86, GM95, MKK97, ZGMHW95, AESY97]. Work on other, mostly on object-oriented, data models has also been done [KR98].

In principle, there are two ways to process updates to views. In the *delta approach*, each update in a source triggers the generation of queries at the view site that serve to update a view extent based on knowledge about the source databases, the view definition, and the updates that have occurred. This approach is very popular in the field of relational databases [BLT86, GMS93, MKK97] but is difficult to implement in XML due to the complex nature of XML updates. There is some initial work available on the WHAX project which attempts to generate such delta-queries [LD00], and partial solutions for some subsets of the problem, involving very simple atomic update operations by [ZM98, AMR⁺98].

The second approach to incremental view maintenance makes use of the fact that queries are translated into algebra expressions to facilitate their execution. One can then define update propagation for each *operator* in the query's algebra tree and, as long as the updatable operators form a closed algebra, one can obtain an algebraic view maintenance solution for queries of arbitrary complexity. This algebra-based view maintenance approach was first proposed for SQL and relational algebra by Griffin and Libkin [GL95] but can be adapted to other, more complex algebras, as demonstrated by Koeller and Rundensteiner [KR02b, KR04]. Jagadish et al. are developing a native XML database system [JAKC⁺02] TIMBER based on an algebra called TAX [JLST02]. However, we are not aware of any work on view update propagation support in the context of TIMBER.

In all of these above cited works, we are not aware of any approaches that investigate a general solution for incremental update propagation in the context of a heterogeneous data models - a key issue when considering the issue of automated to semi-automated maintenance of data sources in the life sciences domain.

8 Conclusions

A fundamental challenge in developing an approach for the incremental propagation of updates from the source to the user's cached data in the life sciences domain is the high variability in the data models. The models of these data sources range from XML to FASTA to ASN.1 to relational. One possible approach, which we quickly discarded, was to develop specific propagation techniques for each pair of data models. Consider for example, the query depicted in Figure 2 to store specific projections of BLAST results (given in FASTA format) into a relational database. This query would require post-query execution (a) parsing of the BLAST results to extract the desired values; (b) mapping (transformation) of these results into relational attributes and tables; and (c) a propagation algorithm to translate the BLAST updates to relational updates based on the specific transformation (step (b)). Such an approach would necessarily need to be modified for different transformations and different data models.

In this paper we present an alternative approach that provides a: (1) a simple yet expressive transformation framework to encompass the steps (a) and (b) described above; and (2) a unique incremental update propagation algorithm based on the transformation framework⁷. The *transformation framework*, based on linear transformations [GY98] is the core of our approach and provides the building blocks necessary to express linear transformations between schemas and data irrespective of their native data model. As a testament to its expressive power, we present two complete, yet diverse examples in this paper. The first example shows how our approach can be applied to transform and subsequently store the pertinent elements from a single source, in this case BLAST. This transformation can be extended with little or no effort by predicates that, for example, filter out all BLAST results below a certain score (`Hsp_score`). The second example, detailed in the case study, focuses on transforming the data from two sources and joining their results in a single target graph.

Beyond its application in the life sciences domain, one of the unique contributions of this work is bringing exposure to the fact that unlike incremental view propagation, propagation at the transformation model level can result in modification of the model itself. We believe this fundamental difference arises from the fact that relational views deal with sets of data while the transformations are dealing with translations typically defined at a lower granularity.

Current Status and Future Work. The fundamental building blocks of the Sangam system together with the incremental update propagation algorithm have

⁷ The transformation framework provides the semantics essential for correct propagation of updates.

been developed and are available for download at <http://www.cs.uml.edu/~kajal/Sangam>. However, we currently do not have a visual component that would allow users to construct their mappings using graphical drag and drop tools. A visual tool is under development and we expect to deploy the system as a set of web services in the early part of next year. A possible direction for future work is to extend the Sangam framework to provide support for higher order transformations. One example of such a transformation is the mapping of an edge between two nodes to a node in the target.

References

- [AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.
- [AGM⁺90] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
- [AMR⁺98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Int. Conference on VLDB*, pages 38–49, August 1998.
- [AT96] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In Peter M. G. Apers et al., editors, *Proceedings of International Conference on Extending Database Technology (EDBT)*, LNCS. Springer, 1996.
- [BA00] A. Bairoch and R. Apweiler. The SWISS-PROT Protein Sequence Databank and its Supplement TrEMBL. *Nucleic Acid Res.*, 1(28):45–48, 2000.
- [BBB⁺98] P.G. Baker, A. Brass, S. Bechhofer, C. Goble, N. Paton, and R. Stevens. TAM-BIS: Transparent Access to Multiple Bioinformatics Information Sources: An Overview. In *Proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology, ISMB98*, 1998.
- [BDH03] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. On the updatability of xml views over relational databases. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, pages 31–36, San Diego, CA, June 2003.
- [BGH⁺00] B.C. Barker, J.S. Garavelli, H. Huang, P.B. McGarvey, B. Orcutt, G.Y. Srinivasarao, C. Xiao, L.S. Yeh, R.S. Ledley, J.F. Janda, F. Pfeiffer, H.W. Mewes, A. Tsugita, and C. Wu. The protein information resource (pir). *Nucleic Acids Research*, 28(1):41–44, 2000.
- [BK02] F. Bry and P. Kroger. A Computational Biology Database Digest: Data, Data Analysis, and Data Management. *International Journal on Distributed and Parallel Databases, special issue on Bioinformatics (to appear)*, 2002.
- [BLT86] J. A. Blakeley, P.-E. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *Proceedings of SIGMOD*, pages 61–71, 1986.
- [BR00] P. A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *International Conference on Conceptual Modeling*, 2000.
- [CAM02] Gregory Cobena, Serge Abiteboul, and Amelia Marian. Detecting changes in XML documents. In *Proceedings of ICDE*, pages 41–52, San Jose, California, February 2002. IEEE.
- [CAW98] S. 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, February 1998.

- [CJR98] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Proceedings of International Conference on Information and Knowledge Management*, pages 314–321, November 1998.
- [Cla02] K.T. Claypool. *Managing Change in Databases*. PhD thesis, Worcester Polytechnic Institute, May 2002.
- [CR03a] Kajal T. Claypool and Elke A. Rundensteiner. Sangam: A framework for modeling heterogeneous database transformations. In *Proceedings of Intl. Conf. on Enterprise Information Systems (ICEIS)*, pages 219–224, Angers, France, April 2003.
- [CR03b] K.T. Claypool and E.A. Rundensteiner. Sangam: A Transformation Modeling Framework. In *DASFAA, Kyoto, Japan, 2003*.
- [CR04] Kajal T. Claypool and Elke A. Rundensteiner. AUP: Adaptive Change Propagation Across Data Model Boundaries. In *Proceedings of 21st British National Conference on Databases (BNCOD)*, pages 72–83, Edinburgh, Scotland, July 2004.
- [CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of SIGMOD*, volume 25 (2) of *ACM SIGMOD Record*, pages 493–504, June 1996.
- [CRZ⁺01] K.T. Claypool, E.A. Rundensteiner, X. Zhang, H. Su, H. Kuno, W-C. Lee, and G. Mitchell. SANGAM: A Solution to Support Multiple Data Models, Their Mappings and Maintenance. In *Demo Session Proceedings of SIGMOD'01*, 2001.
- [CTZ01] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient management of multiversion documents by object referencing. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 291–300, Orlando, September 2001. Morgan Kaufman.
- [EBI02] European Bioinformatics Institute. <http://www.ebi.ac.uk>, 2002.
- [EUA96] T. Etzold, A. Ulyanov, and P. Argos. SRS: Information Retrieval System, for Molecular Biology Data Banks. *Meth. Enzymol.*, 266:114–128, 1996.
- [FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data Using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.
- [GB95] A. Gupta and J.A. Blakeley. Using Partial Information to Update Materialized Views. *Information Systems*, 20(8):641–662, 1995.
- [GDB] GDB. GDB: The Genome Database. <http://gdbwww.gdb.org/>.
- [GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.
- [GL98] S. Göbel and K. Lutze. Development of Meta Databases for Geospatial Data in the WWW. In *ACM-GIS*, pages 94–99, 1998.
- [GM95] A. Gupta and I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Warehousing*, 18(2):3–19, 1995.
- [GMHI⁺95] Héctor García-Molina, Joachim Hammer, Kelly Ireland, et al. Integrating and accessing heterogeneous information sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.
- [GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.
- [GY98] J. Gross and J. Yellen. *Graph Theory and its Applications*. CRC Press, 1998.

- [HKR⁺00] L.M. Haas, P. Kodali, J.E. Rice, P.M. Schwarz, and W.C. Swope. Integrating Life Sciences Data - With a Litte Garlic. In *IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE)*, pages 5–12. ACM Press, 2000.
- [HMN⁺99] L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P. Schwarz, and E.L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.
- [HN99] Eric N. Hanson and Lloyd Noronha. Timer-driven database triggers and alerters: Semantics and a challenge. *SIGMOD Record*, 28(4):11–16, 1999.
- [JAKC⁺02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, et al. TIMBER: A native XML database. *VLDB Journal: Very Large Data Bases*, 11(4):274–291, December 2002.
- [JLST02] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. *Lecture Notes in Computer Science*, 2397:149–164, 2002.
- [Kel82] A. Keller. Updates to Relational Database Through Views Involving Joins. In *Scheuermann*, 1982.
- [KR98] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in MultiView: Strategies and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(5):768–792, Sep./Oct. 1998.
- [KR02a] A. Koeller and E.A. Rundensteiner. Incremental Maintenance of Schema-Structuring Views. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, page to appear, 2002.
- [KR02b] Andreas Koeller and Elke A. Rundensteiner. Incremental maintenance of schema-structuring views. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 354–371, 2002.
- [KR04] Andreas Koeller and Elke A. Rundensteiner. Incremental maintenance of schema-structuring views in SchemaSQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2004. to appear.
- [LD00] Hartmut Liefke and Susan B. Davidson. View maintenance for hierarchical semistructured data. In *Proceedings of 2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, Lecture Notes in Computer Science 1874, pages 114–125, Greenwich, UK, September 2000. Springer Verlag.
- [MACM01] Amélie Marian, Serge Abiteboul, Grégory Cobéna, and Laurent Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 581–590, Orlando, September 2001.
- [MIR93] R. J. Miller, Y. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases (VLDB)*, pages 120–133, Dublin, Ireland, August 1993.
- [MKK97] Mukesh K. Mohania, Shin’ichi Konomi, and Yahiko Kambayashi. Incremental Maintenance of Materialized Views. In *Database and Expert Systems Applications (DEXA)*, pages 551–560, 1997.
- [MR83] L. Mark and N. Roussopoulos. Integration of Data, Schema and Meta-Schema in the Context of Self-Documenting Data Models. In Carl G. Davis, Sushil Jajodia, Peter A. Ng, and Raymond T. Yeh, editors, *Proceedings of the 3rd Int. Conf. on Entity-Relationship Approach (ER’83)*, pages 585–602. North Holland, 1983.

- [MRB03] S. Melnik, E. Rahm, and P. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *Proceedings of SIGMOD*, pages 193–204, 2003.
- [MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *International Conference on Very Large Data Bases*, pages 122–133, 1998.
- [NACP01] Benjamin Nguyen, Serge Abiteboul, Grégory Cobena, and Mihaí Preda. Monitoring XML data on the Web. In *Proceedings of SIGMOD*, pages 437–448, 2001.
- [PDB03] PDB Team. *The Protein DataBank*. John Wiley and Sons, 2003.
- [PDB04] PDBj Home Page: <http://www.pdbj.org>, 2004.
- [PGMW95] Y. Papakonstantinou, Héctor García-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of IEEE International Conference on Data Engineering*, pages 251–260, March 1995.
- [PR95] M.P. Papazoglou and N. Russell. A Semantic Meta-Modeling Approach to Schema Transformation. In *CIKM '95*, pages 113–121. ACM, 1995.
- [RR87] A. Rosenthal and D. Reiner. Theoretically Sound Transformations for Practical Database Design. In Salvatore T. March, editor, *Entity-Relationship Approach, Proceedings of the Sixth International Conference on Entity-Relationship Approach, New York, USA, November 9-11, 1987*, pages 115–131, 1987.
- [SIB04] Swiss-Prot Release 42.11 Statistics.
<http://au.expasy.org/sprot/relnotes/relstat.html>, 2004.
- [SPT04] Uniprot Documents:
<http://www.ebi.uniprot.org/support/docs/uniprot.xsd>, 2004.
- [TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *Proceedings of SIGMOD*, pages 413–424, New York, NY 10036, USA, 2001. ACM Press.
- [Uni02] UniProt—the universal protein resource. <http://www.uniprot.org>, 2002.
- [W3C01] W3C. XQuery: A Query Language for XML.
<http://www.w3.org/TR/xquery/>, February 2001.
- [WDC03] Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings of ICDE*, pages 519–530, Bangalore, India, March 2003. IEEE.
- [WZ03] Fusheng Wang and Carlo Zaniolo. Temporal queries in XML document archives and web warehouses. In *Proceedings of 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic*, pages 47–55, Cairns, Queensland, Australia, July 2003. IEEE.
- [XWW⁺02] Haiyuan Xu, Quanyuan Wu, Huaimin Wang, Guogui Yang, and Yan Jia. KF-Diff+: Highly efficient change detection algorithm for XML documents. In R. Meersman, Z. Tari, et al., editors, *Confederated International Conferences CoopIS, DOA, and ODBASE 2002. Proceedings*, Lecture Notes in Computer Science 2519, pages 1273–1286. Springer Verlag, 2002.
- [ZGMHW95] Y. Zhuge, Héctor García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

- [ZM98] Y. Zhuge and H. Garcia Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th International Conference on Data Engineering, Orlando, Florida*, pages 116–125, February 1998.