Bulletin of the Technical Committee on

Data Engineering

March 1999 Vol. 22 No. 1 **IEEE Computer Society**

Letters

Special Issue on Data Transformations

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering (http: www. is open to all current members of the IEEE Computer Society who are interested in database systems.

The web page for the Data Engineering Bulletin is http://www.research.microsoft.com/research/db/debull. The web page for the TC on Data Engineering is http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the Editor-in-Chief

Our First Electronic "Mostly" Issue

Many TC's of the IEEE have electronic-only distribution of their news letters. We have never thought of the Bulletin as a news letter. Indeed, there is no news section within the pages of the Bulletin except for the occasional news provided in my letters. Rather, the Bulletin, while an informal publication, is primarily a vehicle for disseminating technical information about both research and industrial practice. Most of us are still accustomed to hardcopy distribution of such publications. However, the world is changing, and, with an extra push because of our very limited finances, we with this issue make the leap into the world of electronic media. You will no longer receive hardcopy of the Bulletin via ordinary mail. After providing several alerts on the subject, we now expect that members of the TCDE will access the Bulletin through our web site at

http://www.research.microsoft.com/research/db/debull.

Because many of you may have waited for the hardcopy issue, rather than accessing the web site, we anticipate increased traffic at the Bulletin web site. Please let us know when you experience difficulties. Also, please send us suggestions as to how we can improve the electronic delivery process. Finally, if you are unable to access the Bulletin via our web site and want to continue receiving hardcopy, contact *Tracy Woods, IEEE Computer Society, 1730 Massachusetts Ave., Washington, DC 20036.* (Email: twoods@computer.org)

Technical Committee Meeting at ICDE'99 in Sydney, Australia

From the TCDE Web Page: All members of the TCDE are invited to the Open Meeting of the TCDE to be held at the ICDE meeting in Sydney Australia from 17:40 to 18:30 on Wednesday March 24, 1999. ICDE Conference details (hotel, registration information, program) can be found at http://www.cse.unsw.edu.au/icde99. The purpose of an open meeting is to inform the TC members what its executive committee has been doing and to give the members a chance to ask questions and provide suggestions. This meeting will be chaired by Prof. Dr. Erich Neuhold, the vice-chair of the TCDE ("E.N.").

The meeting agenda and more complete information can be found on the TCDE web site at

http://www.ccs.neu.edu/groups/IEEE/tcde/.

This Issue

Data transformations have a long history in the database field, and in our reseach literature as well. Early in the evolution of relational databases, this area attracted a substantial amount of research interest. There was then a quiescent period in which researchers largely ignored the field. More recently, however, the emergence of data warehouses has sparked renewed interest. Data transformations, including not only model transformations and other syntactic issues, but also data scrubbing and various forms of semantic transformations became crucial for the successful data warehouse. An added incentive for work in this area came from the web explosion, which led to renewed interest in enterprise and cross-enterprise data integration. Data transformation is critical to most wide-scale integration. Elke Rundensteiner has successfully solicited papers from both researchers and commercial organizations that provide broad coverage of the data transformation field, covering future research possibilities and current or soon to be available practice. Thus the issue is a very nice blend of papers that should serve as a good introduction for members of our technical community. I want to thank Elke for her hard work in bringing this issue to fruition, and for being responsive to unanticipated scheduling constraints.

David Lomet
Microsoft Research

Letter from the Special Issue Editor

Data transformation is the bread and butter technology involved in most aspects of information management. Transformations are needed to migrate legacy systems to more modern applications, to optimize queries, to translate from one data model to another, to integrate heterogeneous systems into federated databases or warehouses, to perform data cleansing or scrubbing, to evolve a schema and its associated database as driven by changing user requirements, to construct user-customized web sites, and to achieve enterprise-wide integration. While the specific data models being transformed have grown over time to include network model, relational data model, object-oriented schema, and now possibly XML web models, the problems of how to specify such mappings, what language to employ, and how to efficiently execute them have persisted. As demonstrated by the large number of authors that contributed to this issue, interest in this technology is as strong as ever. Although there are a variety of products on the market that achieve some forms of transformations, many are limited to a specific problem and/or a specific data model. How to provide a general umbrella approach solving all or at least a large class of the above problems remains unanswered. This issue reports upon recent efforts from academia and industry both on addressing specific transformation problems as well as on developing more generic transformation approaches.

Abiteboul, Cluet, et al. address data translation for heterogeneous sources. Their solution assumes a middleware data model to which data sources are mapped, along with a declarative language for specifying translations within this common middlelayer data model. Automation of some of the tasks of translation are studied.

Bernstein and Bergstraesser report on facilities integrated with Microsoft SQL Server 7.0 for transforming both data and meta-data. They describe the repository functions of the meta-data manager called Microsoft Repository, such as the Data Transformation Service which helps users develop programs to populate a data warehouse.

Carrer, Joshi, et al. describe the Oracle MediaAnnotator, a metadata translation and management tool to be used with Oracle8i interMedia. MediaAnnotator automatically extracts and transforms metadata from multimedia objects into logical annotations, thus simplifying the indexing and searching of multimedia objects.

Claypool and Rundensteiner present a generic framework for flexible database transformations, called SERF. SERF can be added as thin transformation support layer on top of current schema evolution systems, enhancing them with flexibility, extensibility and re-usability of transformations.

Davidson and Kosky introduce a declarative (Horn-clause) language, WOL, developed for specifying transformations involving complex types and recursive data-structures. WOL transformations, which map from a source to a target schema, are easily modified to respond to schema evolution of the source schema. WOL mappings can explicitly resolve incompatibilities between source and target.

Haas, Miller, et al. advocate database middleware systems as transformation engines. They argue that a middleware transformer must provide database-like features, in particular, transformation support for heterogeneous data and efficient query processing capabilities to compensate for less-capable sources.

Heiler, Lee, and Mitchell apply software repository technology to the problem of legacy system migration. Correctly transforming systems (including data, programs and processes) requires metadata about the two objects and their interrelationships. Hence, metadata repositories that provide tools for capturing, transforming, storing, and manipulating metadata are advocated for supporting the migration process.

Hellerstein, Stonebraker, and Caccia introduce the Cohera Federated DBMS as a way to add data independence back into the process of integrating heterogeneous databases. They advocate physical independence for scalable physical design of enterprise-wide data, and logical independence via the use of SQL99 to achieve an open conversion framework.

Finally, Orenstein describes the object/relational mapping system called Enterprise Business Objects (EBO) which is part of a Java application server. Transformations used by EBO to provide transparent, high-performance database access to middle-tier Java applications are characterized.

Elke A. Rundensteiner
Worcester Polytechnic Insitute

Tools for Data Translation and Integration *

Serge Abiteboul† Sophie Cluet† Tova Milo‡ Pini Mogilevsky‡ Jerome Siméon†
Sagit Zohar‡

Abstract

A broad spectrum of data is available on the Web in distinct heterogeneous sources, stored under different formats. As the number of systems that utilize this data grows, the importance of data conversion mechanisms increases greatly. We present here an overview of a French-Israeli research project aimed at developing tools to simplify the intricate task of data translation. The solution is based on a middleware data model *to which various data sources are mapped, and a* declarative language *for specifying translations within the middleware model. A complementary schema-based mechanism is used to automate some of the translation. Some particular aspects of the solution are detailed in [3, 7, 10].*

1 Introduction

Data integration and translation is a problem facing many organizations that wish to utilize Web data. A broad spectrum of data is available on the Web in distinct heterogeneous sources, stored under different formats: a specific database vendor format, XML or Latex (documents), DX formats (scientific data), Step (CAD/CAM data), etc. Their integration is a very active field of research and development, (see for instance, for a very small sample, [4, 5, 9, 6]). A key observation is that, often, the application programs used by organizations can only handle data of a specific format. (e.g. Web browsers, like Netscape, expect files in HTML format, and relational databases expect relations). To enable a specific tool to manipulate data coming from various sources (e.g. use, in a relational system, data stored on the Web in HTML format), a translation phase must take place – the data (in the source format) needs to be mapped to the format expected by the application.

The naive way to translate data from one format to another is writing a specific program for each translation task. Examples are the Latex to HTML and HTML to text translators [8, 1]. Writing such a program is typically a non trivial task, often complicated by numerous technical aspects of the specific data sources that are not really relevant to the translation process (e.g. HTML/XML parsing, or specific DB access protocol). A sound solution for a data integration task requires a clean abstraction of the different formats in which data are stored, and means for specifying the correspondences between data in different worlds and for translating data from one world to another. For that, we introduced a *middleware* data model that serves as a basis for the integration task, and *declarative rule languages* for specifying the integration. Translation from source to target formats is achieved by (1) importing data from the sources to the middleware model, (2) translating it to another middleware representation that better fits the target structure, and then (3) exporting the translated data to the target system.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* Work partially supported by AFIRST and the Israeli Ministry of Science.
† I.N.R.I.A, Rocquencourt, France Email: <firstname>.<lastname>@inria.fr
‡ Tel Aviv University, Israel, Email: {milo,pinim,sagit}@math.tau.ac.il

A first attempt: Our first attempt (to be refined later on) was to propose a very simple middleware model where data is represented by ordered labeled trees. The model is very similar to the common OEM model for semistructured data. A particularity here is that we allow an order to be defined on the children of nodes. Order is an inherent component of some data structures, e.g. ordered tuples and lists. Similarly, textual data can either be described as a sequence of characters or words, or on a higher level by a certain parse tree; in both cases, the order of data element is important. Supporting order as part of the data model enables a natural representation of data coming from such sources [3]. Together with the data model, we introduced a declarative rule-based language for describing correspondences between data in different worlds [3]. The language has several useful properties: it provides constructs to search and build complex trees that represent data; the semantics of rules take into consideration the fact that nodes may represent collections with specific properties (e.g., sets are insensitive to order and duplicates); the complexity of correspondence computation was proved to be PTIME in many practical cases. Of course, deriving correspondences within existing data is only one issue in a heterogeneous context. One would also want to translate data from one representation to another. Interestingly, we showed that in many practical cases, translation rules can automatically be derived from the correspondence rules. Thus, a complete integration task (derivation of correspondences, transformation of data from one world to the other, incremental integration of a new bulk of data, etc.) can be specified using a *single* set of declarative rules. This was an important result. It simplifies the specification task and helps in preventing inconsistencies in specifications.

To experiment with the above translation scheme we built a prototype system, W3TRANS [11], and used it to define mapping and translations among various formats including SGML, HTML, relational and OODBs. We soon realized that although this simplified greatly the translation specification, still, very often, a considerable programming effort was required when a new translation was to be defined. Our goal was thus to design additional mechanisms to further simplify the translation. The solution was based on two main observations.

(i) Instantiation: In many cases, the translations needed by the user are combinations or specialization of other translations. For example consider the following application. A research institute wants to build an intranet application. Among other things, the institute stores informations about its researchers in an OODB, and keeps their research papers in XML documents. Assume we want to provide an HTML interface to the system so that employees can view it on the Web. If the system happens to contain two generic programs providing an HTML view of OO and XML data resp., then the specific application can be obtained by customizing and combining the above programs. Following this idea we enhanced our model by a novel *instantiation* mechanism, extending both the data model and the rule-based translation language and providing a powerful mechanism for customization and synthesis of programs. Interestingly, both the model and the rules have a natural graphical representation which further simplifies the programming by allowing a convenient graphical interface via which the user can, with minimal programming, customize and combine existing programs.

(ii) Schema-based translation: Frequently, much of the structure of the source data is similar to that of the target translated data, and many of the structure modifications to be performed by the translation process are rather standard and result from various differences between the schemas of the source and target systems. We use here the general term *schema* to denote whatever way a data model chooses to model its data. For example, databases use schemas to model database instances; Structured documents often obey some grammar (e.g. Document Type Definition – DTD – in SGML and XML); In other models such a definition may be partial (e.g. in semistructured data [2]). The observation is that, in many translations, the schema of the target system is closely related to that of the source system – both schemas aim to represent the same data. This implies that a large part of the translation can be done *automatically*, relying on this (often standard) relationship, hence reducing the programming effort, and involving the user only in the specification of the "non standard" parts of the translation.

Based on the above observations we have developed two complementary prototype systems, YAT and TranScm. YAT focuses on the rule language and implements the instantiation and customization mechanisms. TranScm focuses on schema analysis and automatic generation of translations when possible. Both support a convenient graphical interface and together provide a sound solution to the data translation problem. In the rest of this

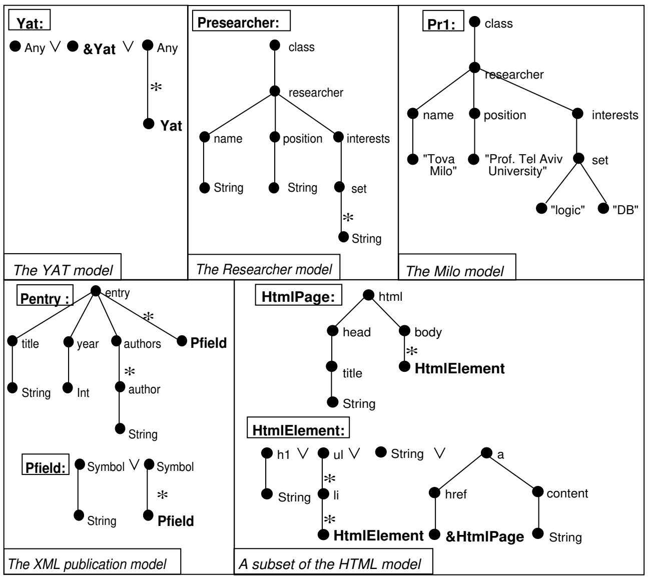

Figure 1: Some YAT Models

paper we briefly describe the main ideas underlying the two systems. For lack of space the presentation is rather informal and is mainly based on examples and intuitive explanations. For full details see [7, 10].

2 YAT

We explain below the main ideas using a running example - the research institute repository mentioned above.

The data model: The YAT data model consists of named ordered labeled trees that may be unioned (with the $\vee$ symbol) to form a pattern. A set of patterns form a model, which is used to represent real-world data (OODB, XML and HTML in our example). The most interesting feature of YAT is that it allows to represent data at various levels of detail. For instance, Figure 1 shows several such models. The three models on the upper part of the figure may be used to represent an OODB object corresponding to the researcher whose name is Tova Milo. Going from left to right, the models get more and more precise. The first one allows to represent any YAT compliant data. The *Researcher* model represents an OODB class *researcher* but also all objects belonging to that class. The last model is an actual representation of the object itself like in usual semistructured data models. Each of these three models is an instance of its predecessor(s). Another interesting example is the XML model in the lower left hand side of the figure. It mixes precise with non-specific informations. An XML entry is described as being a sequence of fields. The first fields are precisely described (i.e., $title$, $year$, $authors$). Then, the pattern just indicates that they are followed by zero or more fields. Each field has a label of type $Symbol$ and may be composed of a simple string or of a sequence of other fields. Finally, the last model (partially) represents any HTML data (internal nodes, e.g. $h1$, ul, a, represent HTML tags).

Let us now see how these models are constructed and give an intuition of the instantiation mechanism. The nodes of a YAT tree may be labeled with a (i) constant (e.g., "Tova Milo"), (ii) symbol (e.g., $class$, $researcher$, set), (iii) type (e.g., Any, $String$, $Symbol$), (iv) pattern name (e.g., $\boldsymbol{Yat}$) or (v) reference to a pattern name (e.g., $\&\boldsymbol{Yat}$). The edges of a YAT tree may be annotated by an occurrence symbol. In the example, there is only one such symbol: $\star$. It indicates the possibility to replace the annotated edge by an arbitrary number of edges (e.g., the two set rooted subtrees).

A model is an instance of another, if one can find a mapping between their edges and nodes. An edge is instantiated according to its occurrence symbol. A node is instantiated according to the following rules:

$\boldsymbol{HtmlPageAll}()$ **:**
$html\langle \to head \to title \to$ "Verso Researchers",
$\quad \to body\langle \to ul \stackrel{\star}{\to} li \to a$
$\qquad\qquad \langle \to href \to \&\boldsymbol{HtmlPage}(N),$
$\qquad\qquad\quad \to content \to N\rangle\rangle\rangle$
$\Longleftarrow$
$\quad \boldsymbol{Presearcher}$ **:**
$\quad class \to researcher$
$\qquad\qquad \langle \to name \to N,$
$\qquad\qquad\quad \to position \to P,$
$\qquad\qquad\quad \to interests \to set \stackrel{\star}{\to} I\rangle$

Figure 2: Rule **All Researchers**

$\boldsymbol{HtmlPage}(N)$ **:**
$html\langle \to head \to title \to H1,$
$\quad \to body\langle \to h1 \to H1,$
$\qquad\quad \to$ "Position is ",
$\qquad\quad \to P$
$\qquad\quad \to ul \stackrel{[]_T}{\to} li\langle \to$ "title : ",
$\qquad\qquad\qquad \to T,$
$\qquad\qquad\qquad \to$ "published in ",
$\qquad\qquad\qquad \to Y\rangle\rangle\rangle$
$\Longleftarrow$
$\quad \boldsymbol{Pentry}$ **:**
$\quad entry\langle \to title \to T,$
$\qquad\quad \to year \to Y,$
$\qquad\quad \to authors \stackrel{\star}{\to} set \to N,$
$\qquad\quad \stackrel{\star}{\to} \boldsymbol{Pfield}\rangle,$
$\quad \boldsymbol{Presearcher}$ **:**
$\quad class \to researcher$
$\qquad \langle \to name \to N,$
$\qquad\quad \to position \to P,$
$\qquad\quad \to interests \to set \stackrel{\star}{\to} I\rangle,$
$\quad H1$ `is` concat("Homepage of ", N)

Figure 3: Rule **Single Researcher**

- A constant/symbol node can be instantiated by nodes with an identical label (e.g., the nodes labeled $class$).
- A type node can be instantiated by a node whose label is a subtype or an instance of that type. E.g., the node labeled Any can be instantiated by the nodes labeled $String$, $Symbol$, "Tova Milo" or $class$.
- A pattern node can be instantiated by a tree, instance of that pattern. E.g., the two trees whose roots are labeled $researcher$ are instances of the $\boldsymbol{Yat}$ labeled node.
- A referenced pattern node can be instantiated by a node whose label is a reference to an instance of that pattern. E.g. the $\&\boldsymbol{HtmlPage}$ labeled node is instance of the $\&\boldsymbol{Yat}$ labeled node.

Defining translations: Now that we are able to describe the structural input (OODB and XML) and output (HTML) of our application, we need a way to map the former to the later. This is done using the translation language YATL (either directly or through its graphical interface) that allows to describe complex conversions on graphs. It is rule-based, each rule describing a small part of the data conversion. For space reasons we will not give the complete program allowing to map researcher objects and XML documents to HTML but two simplified, yet significant, rules. Although, the syntax may look somewhat complicated, note that: (i) most of the time, the programmer only has to modify some rules already existing in the system and, from our experience, can master the syntax of the rules rather fast; (ii) the programmer has the option to use the graphical interface to modify such rules [7]. We now illustrate the YATL language on two simplified rules of the example application.

Consider the rule **All Researchers** on Figure 2. This rule creates a single Web page that contains a list of pointers to all researchers in the database. On the lower part of the rule (i.e., after the $\Longleftarrow$ symbol), the pattern $\boldsymbol{Presearcher}$ is used to filter the input data and retrieve the required information through variable bindings (in the example, the researchers names are bound to Variable N). On the upper part, the output pattern describes how the HTML structure is generated. The output of the rule is identified by a Skolem function: $\boldsymbol{HtmlPageAll}()$.

The fact that this Skolem function has no parameter indicates that only one pattern will be generated, including the transformation of all the input filtered data. The head of the rule contains a reference to another Skolem function, this time parameterized by the researchers names ($\&\boldsymbol{HtmlPage}(N)$). Note also that we could have used the researcher's identity as a parameter, replacing N by $\boldsymbol{Presearcher}$ in the Skolem function. As it is, there will be one reference per researcher name. The HTML page associated to this reference is described by another rule **Single Researcher** given in Figure 3.

This rule computes the page associated to one researcher identified by its name. It integrates data from the OODB and the XML files. Its input is a set of patterns, instances of the $\boldsymbol{Pentry}$ and the $\boldsymbol{Presearcher}$ patterns. Its output is a new HTML page identified by the $\boldsymbol{HtmlPage}(N)$ Skolem function. Remark that Variable N is used in both $\boldsymbol{Pentry}$ and $\boldsymbol{Presearcher}$ patterns, making the correspondence between a given researcher in the object database and its publications in the XML file. The page title $H1$ is generated using a function that concatenates two string. Among other things, the page contains a list of the researcher publications (ul), which is ordered by title. This is specified by the $\stackrel{[]_T}{\rightarrow}$ edge which states that a new child is created for each distinct title T (grouping) and that the children must be ordered according to the value of T (ordering). The ordering possibilities of YATL are mandatory to manage Web documents, and can be used for example to give several views of a given list of publications (ordered by title, by publication date, by authors, etc).

Just like the data model, translation rules can also be instantiated and customized for specific needs. The programmer can start with a general program that can be instantiated into one that corresponds to the given input. Customization can follow. For example, through successive instantiations/customizations, we can find programs generating HTML pages from (i) any input, (ii) an arbitrary OODB (iii) data corresponding to a specific OODB schema or (iv) an object of that schema. It is also possible to compose programs and efficiently build complex translations out of simple, previously defined blocks. We omit this here for lack of space.

3 TranScm

To further simplify the programming, we observe that in many cases the schema of the data in the source system is very similar to that of the target system. In such cases, much of the translation work can be done automatically, based on the schemas similarity. This can save a lot of effort for the user, limiting the amount of programming needed. The TranScm system implements the above idea. Given the schemas for the source and target data, the system examines the schemas and tries to find similarities/differences between them. This is done using a rule-based method, where each rule (1) defines a possible common matching between two schema components, and (2) provides means for translating an instance of the first to an instance of the second. The system has a set of build-in rules that handles most common cases, and that can be extended/adjusted/overridden by the user during the translation process. The system uses the rules and tries to find for each component of the source schema a unique "best matching" component in the target schema, or determine that the component should not be represented in the target. This is called the *matching process*. If the process succeeds, the data translation can be performed automatically using the translation facilities of the matching rules. There are two cases where the process may fail: (i) a component of the source schema cannot be matched with a target one using the current set of rules, (and the matching process can neither derive that the component should be just ignored), or (ii) a component of the source schema matches several components in the target schema, and the system cannot automatically determine which is the "best" match. For (i) the user can add rules to the system to handle the special component and describe the translation to be applied to it. For (ii) the user is asked to determine the best match. Then, based on the user's input, the matching process is completed and the translation is enabled.

Handling schemas of different sources requires a common framework in which they can be presented and compared. Note however that the middleware model presented above, with its instantiation mechanism, allows to present not only data but also schema, hence can serve as such common framework. (In the actual implementation of the system we used a slightly more refined model, but the differences are irrelevant for the purpose of the current paper). Similarly, the match&translate rules used by the system could be defined in the style of YATL. (In fact, the system provides a generic interface for rules, independent of the specific language used to specify them. This enabled us for example to use in the prototype Java as the rule definition language.)

A typical scenario of the system's work is as follows. It receives as input two schemas, one of the data source and the other of the target. The two schemas are imported into the system and presented in the common schema model. The next step is matching. The system tries to find for every component in the source schema a corresponding component in the target schema (or determine that the component should not be represented in the target), using the rule-based process mentioned above. Once the matching is completed (perhaps with the user's assistance), a data translation is possible. To perform the translation, a data instance of the source schema is imported to the common data model, and is "typed", i.e. every element in the data is attached a corresponding schema element as a type. Now, the system uses the match between the schema components, achieved in the previous step, to translate the data. Recall, from above, that every rule in the system has two components, the first defines a possible common matching between two components of schemas, and the second provides means for translating an instance of the first to an instance of the second. Every element of the source data is translated, using the translation function of the rule that matched its type with a type (component) of the target schema, to an instance of the target type. The resulting elements are "glued" together to form a valid instance of the target schema. Finally, the translated data is exported to the target application.

To conclude, note that the schema-based translation method is not aimed at *replacing* the YATL programming language, but rather to *complement* it - rather than writing a translation program for all the data, much of the translation will be done automatically by the system, based on the schema matching, and the programmer needs to supply only some minimal additional code handling the data components not covered by the system.

References

[1] The html2text translator, 1996. Available by from `http://www.scanline.com/html2txt/`.

[2] S. Abiteboul. Querying semi-structured data. In *Proc. ICDT 97*, pages 1–18, 1997.

[3] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proc. ICDT 97*, pages 351–363, 1997.

[4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 505–516, 1996.

[5] M.J. Carey et al. Towards heterogeneous multimedia information systems : The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[6] 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 IPSJ Conference*, pages 7–18, Tokyo, Japan, October 1994.

[7] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 177–188, 1998.

[8] Nikos Drakos. The latex2html translator, 1996.
Available from `cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html.ps`.

[9] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 251–262, 1996.

[10] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 122–133, 1998.

[11] Pini Mogilevsky. W3trans translation system, 1996.
Available from `www.math.tau.ac.il/pinim/w3trans-home.html`.

Meta-Data Support for Data Transformations Using Microsoft Repository

Philip A. Bernstein Thomas Bergstraesser
Microsoft Corporation
[philbe, thomberg]@microsoft.com

Abstract

Data warehousing requires facilities for moving and transforming data and meta-data. This paper describes such facilities that are integrated with Microsoft SQL Server 7.0, particularly as they relate to meta-data and information models in its shared repository. It also discusses the use of XML to move meta-data between tools.

1 Introduction

To interpret data, users and applications need meta-data. Therefore, whenever data is moved or transformed, meta-data needs to be involved. Meta-data controls the movement by specifying such information as location, schedule, and source-to-target mappings. It needs to be moved along with the data so that further work can be done on the data after the move. Meta-data that describes the data movement activity is also useful to retrace data movement steps later on.

Like any kind of data, meta-data requires a persistent data store (i.e., a repository) in which to keep it [1]. It also requires an information model (i.e., schema) that describes the meta-data to be managed. The information model supports the integration of relevant tools and enables sharing of meta-data through the repository.

Today, the most common application environment in which data is moved and transformed is the data warehouse, an increasingly important part of an enterprise's information architecture. The amount and diversity of meta-data required in a data warehouse makes it a natural application of repository technology. Schema management for source and target data sources, aggregation information, scheduling of tasks, and system topology are examples of meta-data that needs to be managed.

Data warehousing scenarios are major applications of Microsoft[1] Repository, a meta-data manager that ships with Microsoft SQL Server 7.0. It includes a repository engine that sits on top of Microsoft SQL Server and supports both navigational access via object-oriented interfaces and SQL access to the underlying store. Microsoft Repository also includes the Open Information Model, a repository schema that covers the kinds of meta-data that are needed for data warehouse scenarios. One of the many users of the repository is SQL Server's Data Transformation Service (DTS), a tool that helps users design programs to populate a data warehouse.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] ActiveX, Microsoft and Visual Basic are trademarks of Microsoft Corporation.

This paper describes facilities that support the movement and transformation of data and meta-data, particularly as they relate to the repository functions that support them. Section 2 describes data warehouse scenarios in more detail, and how DTS addresses them. Section 3 discusses the problem of sharing and exchanging meta-data via XML. Section 4 summarizes some future directions.

The first release of Microsoft Repository and the Open Information Model was in early 1997 in Visual Basic 5.0 [2]. The second release, which includes version and workspace management and a much expanded information model, shipped in Visual Studio 6.0 and SQL Server 7.0 in late 1998 [3]. The product also has a software development kit (SDK) that includes tools for designing and extending information models. Details are available at [7].

2 Populating a Data Warehouse

The most labor-intensive parts of creating a data warehouse are cleansing and mapping. Data from many sources is moved through a sometimes complex scrubbing process whose goals are to normalize the representation, filter out the desired information, and preserve information quality in the target system. To make this error prone process traceable, sophisticated systems maintain additional lineage information that links data to meta-data that describes how the data was processed.

Rules must be defined and implemented to identify and resolve inconsistent data formats, missing data, invalid data, semantic inconsistencies, and other sources of poor data quality. Often, data must be merged from multiple data sources and reconciled. Sometimes it is aggregated into summary information. Even a relatively simple data warehouse can require a hundred cleaning and mapping rules, with effort measured in person years.

Anecdotal evidence from consultants and users tells us that data scrubbing is over two-thirds of the work in setting up and managing a data warehouse. For this reason, there is a growing market for tools that simplify this work, a market that many software vendors are trying to fill. As explained earlier, these tools need meta-data: schema descriptions, libraries of transformation rules, descriptions of transformation scripts, etc.

Data warehousing is becoming one of the largest segments of the database business. SQL Server in its 7.0 release addresses this segment with an easy-to-use out-of-the-box data mart solution and a platform for scalable data warehouses. The main built-in components are an OLAP engine for accessing multi-dimensional data (Microsoft Decision Support Services) and a data movement tool (Data Transformation Service (DTS)). Meta-data management and extensibility come via Microsoft Repository and the Open Information Model.

DTS is a graphical tool for constructing and running transformation packages. Each package is a workflow that defines a transformation. It consists of a set of partially ordered steps, each of which invokes a task. Such a task can execute SQL statements, call an ActiveX script or external program, perform a bulk insert, send E-mail, transfer whole database objects, or call a data pump.

Data pump tasks are the main workhorse. They retrieve and store data through OLE DB, which defines a standard interface for access to heterogeneous data [4]. Each data pump task copies and/or transforms data from OLE DB data sources to OLE DB data targets. It consists of a set of transformations, each of which maps a set of input tables and columns to a set of output tables and columns. A transformation ranges from a simple copy operation to the invocation of a custom C++ program or ActiveX script.

The DTS Designer is a graphical tool for creating DTS packages (see Figure 1). Package definitions are meta-data and can be stored in a file, in SQL Server, or in Microsoft Repository. The latter is required if a user wants to track package versions, meta-data, and data lineage, all of which are maintained in the repository as fine-grained objects.

DTS's usage of the repository as a meta-data store is based on two main sub-areas of the Open Information Model (OIM): database models and data transformation models. The core database model (DBM) includes the concepts found in SQL data definitions and similar models of formatted data. Thus, it includes the concepts of database catalog, database schema (e.g., all objects owned by a user), column set (an abstraction of table and

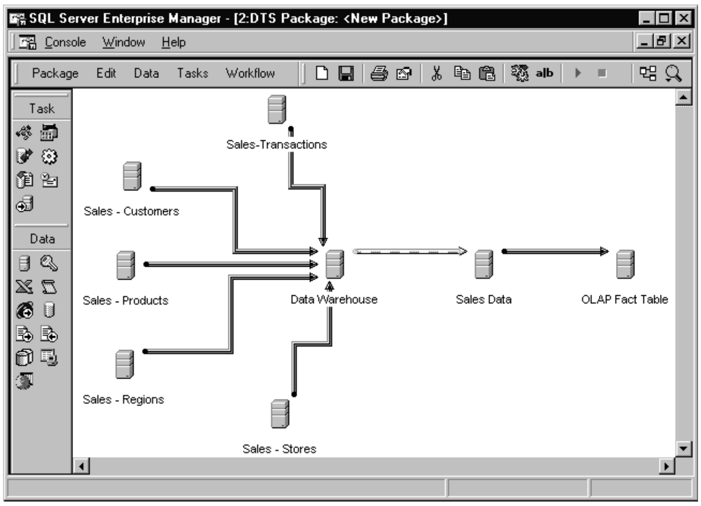

Figure 1: The DTS Designer

view), table, view, constraint, trigger, stored procedure, and column.

OIM is described in the Unified Modeling Language (UML) [5, 6, 8]. It also uses UML as its core model from which sub-models of OIM inherit. Inheriting UML concepts reduces the overall size of the model and promotes sharing between sub-models. For example, a subset of DBM is shown in Figure 2, where gray boxes denote UML core concepts from which DBM concepts inherit: catalog and schema inherit from UML package (UML's general container concept); column set, table, and view inherit from UML type; and column inherits from UML attribute (a structural feature of a UML type). Similarly, constraint inherits from UML constraint and trigger inherits from UML method (not shown in the figure). Some DBM concepts do not have natural analogs in UML, such as key and index, so they inherit from the very abstract UML concept of model element.

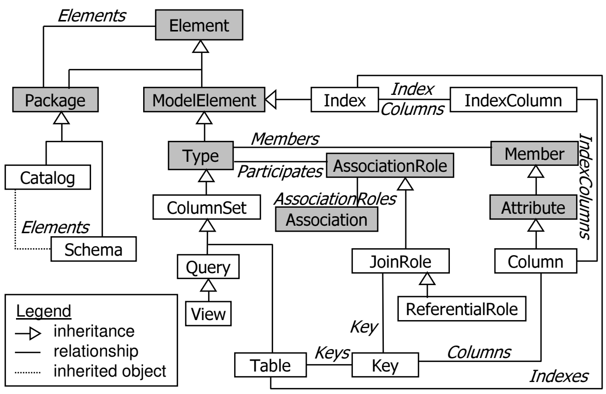

Figure 2: Database Information Model (DBM)

The database information model is populated with schema information by utilities included with SQL Server. The utilities use OLE DB and therefore can import schemas from any OLD DB data source, which include most popular database products (see http://www.microsoft.com/data/oledb/).

The transformation model (TFM) is another sub-model of OIM, which captures information about compound data transformation scripts (see Figure 3). An individual transformation (which is a specialization of UML method) can have relationships to the sources and targets of the transformation. Transformation semantics can be captured by constraints and by code-decode sets for table-driven mappings. Transformations can be grouped into a task (the unit of atomic execution), which is invoked by steps (the unit of scheduling) within a top-level package (which is a specialization of the generic UML package).

To support data lineage, package executions are also modeled. When a package is executed, DTS stores a description of that package execution in the repository. It also optionally tags each data warehouse row by the identifier of the package execution that produced it. This identifier enables traceability from the row in a data table to the repository's description of package execution that populated or updated it. From there, one can trace back to the package transformations, the sources and targets of the transformation, etc., to get a complete description of where that row came from.

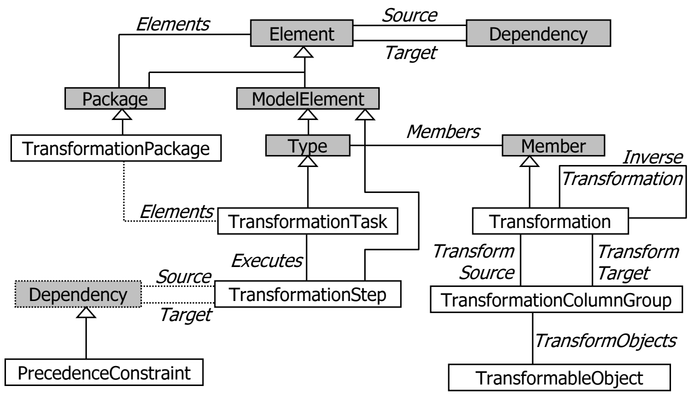

Figure 3: Transformation Model (TFM)

The DBM and DTS models were developed in collaboration with several vendors and reviewed by hundreds of others. Many of these vendors ship products that store or retrieve meta-data that are instances of these models. This enables them to "upsell" to customers whose data warehouse problems outgrow the complexity or scale that Microsoft's framework can handle using the built-in tools.

OIM groups models into generic ones that cover the most common meta-data types in each domain and tool-specific ones that model private data structures. For example, the generic TFM model is specialized to the DTS model, which is a product-specific sub-model for the DTS tool. Primarily, the DTS model captures detailed properties of DTS's specializations of TFM script objects. For example, it includes properties of data pump tasks, transfer object tasks, and send mail tasks, all of which inherit from TFM transformation task, and of transformations, steps, and step executions.

Other sub-models of OIM that are relevant to data warehouse scenarios are the OLAP information model (OLP) for multi-dimensional schemas and Semantic Information Model (SIM) for semantics to drive a natural language query tool. The repository integrates these tools by importing a multi-dimensional schema as OLP objects from Microsoft OLAP Services and semantic information as SIM objects from Microsoft English Query.

3 Using XML

XML (extensible Markup Language) and its family of technologies are standards managed by the World Wide Web Council (W3C) for representing information as structured documents [9]. The structure of a document is expressed by a DTD (Document Type Definition) that consists of tag definitions and rules for how tagged elements can be nested. The basic idea behind XML is to embed tags into a character stream so that semantic structures can be easily recognized. Figure 4 shows an ASCII stream containing the address of a company, a DTD that captures the structure of an address definition, and the ASCII stream with embedded XML tags that conforms to the DTD.

Tagged structures like the one in Figure 4 can be embedded in Web pages to allow more efficient indexing and searching, an important application area for XML. However, there is an equally broad scope for XML in the area of data exchange and transformation. XML will become the standard way to define the structure of ASCII data streams in a world of heterogeneous systems. Agreeing on a simple DTD for the exchange allows systems to interpret ASCII data, to map it onto database tables, or to convert it into different formats.

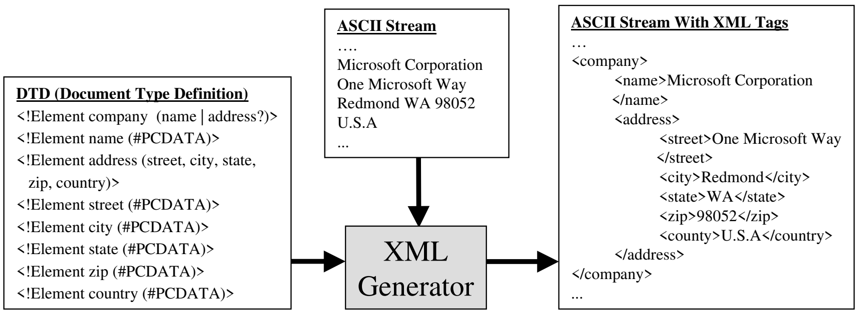

Figure 4: XML Example

It is therefore very natural to use XML to interchange meta-data – highly structured data described by an information model. To use XML to exchange meta-data, one needs to agree on models that describe the semantics and structure of meta-data to be exchanged. OIM is one such model. Because of its formal description in UML, we have been able to automatically map each sub-model of OIM into a DTD that guides the encoding of instances of that sub-model (see Figure 5). The mapping is a set of simple rules for generating DTD statements from the meta-data specifications.

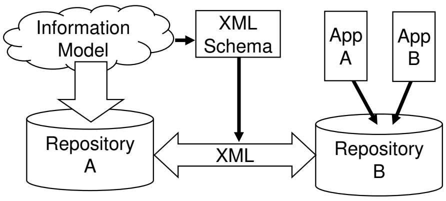

Figure 5: Using OIM XML to Transfer Meta-Data

An example of the OIM XML format is shown in Figure 6. Each element tag includes the OIM sub-model and type within the model. For example, dbm:Table describes the type Table within the database model, DBM. XML import and export programs are in the Microsoft Repository SDK [7]. Using this format, tool vendors can share information in the repository and therefore with DTS, with Microsoft-supplied importers, and each other.

Longer term, we expect unification of data and meta-data transfer via a rich set of efficient XML-based tools. This will strengthen the tool sets currently available for meta-data transfer. And it will simplify data transfer by making accompanying meta-data transfer more readily available.

4 Summary and Futures

Moving and transforming data with its meta-data in an enterprise environment is a complex, error phone, and expensive activity. Data, meta-data that describes data, and meta-data that guides the processing of data are all closely related. They cannot be treated separately if the data and transformation processes are to be manageable and comprehensible.

Microsoft and partner companies have developed the Open Information Model to provide an integration platform for data movement and processing tools. The next step is to extend the model to capture more of the semantics of transformations and to provide a more formal representation of business rules behind transformations.

```
<?xml version="1.0" ?>
<?xml:namespace ns="xif.dtd" prefix="xif" ?>           Sub-model
…
<xif:transfer version="1.1">
    <dbm:Table xif:id="_1">
        <xif:name>Addresses</xif:name>              Property
        <uml:members xif:count="2">
            <dbm:Column xif:id="_2" xif:ordinal="1">
                <xif:name>Zip Code</xif:name>
                <dbm:type>Numeric</dbm:type>
            </dbm:Column>
            <dbm:Column xif:href="#_3" xif:ordinal="2">
            </dbm:Column>
        </uml:members>                      Object Reference
    </dbm:Table>
    …
</xif:transfer>
```

Object

Relationship

Object

Figure 6: Transferring a Table Definition Using XML

This includes a more formal representation of semantic mappings between different models and their encoding, especially in XML.

To broaden the scope of the OIM as a description of meta-data about data, Microsoft and the Meta Data Coalition (MDC), a four-year-old industry consortium devoted to meta-data standards, have recently announced the forthcoming transfer of OIM to the MDC [7]. The MDC will maintain and evolve the OIM from its current COM orientation into a technology-independent and vendor-neutral meta-data standard. This will allow repository and tool vendors to use the OIM with their products independent of Microsoft and, combined with XML, move this meta-data between their products

References

[1] Bernstein, P.A. "Repositories and Object-Oriented Databases" In *Proceedings of BTW '97*, Springer, pp. 34-46 (1997). (Reprinted in *ACM SIGMOD Record 27, 1* (March 1998)).

[2] Bernstein, P.A., B. Harry, P.J. Sanders, D. Shutt, J. Zander, "The Microsoft Repository," *Proc. of 23rd Int'l Conf. on Very Large Data Bases*, Morgan Kaufmann Publishers, 1997, pp. 3-12.

[3] Bernstein, P.A., T. Bergstraesser, J. Carlson, S. Pal, P.J. Sanders, D. Shutt, "Microsoft Repository Version 2 and the Open Information Model," *Information Systems 24(2)*, 1999, to appear.

[4] Blakeley, J., "Data Access for the Masses through OLE DB," *Proc. 1996 ACM SIGMOD Conf.*, pp. 161-172.

[5] Booch, G., J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

[6] Fowler, M., and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 1997.

[7] Microsoft Corp., Microsoft Repository web site, http://www.microsoft.com/repository

[8] Object Management Group, OMG Technical Library, at http://www.omg.org.

[9] World Wide Web Consortium, XML Language Specification and related documents, at http://www.w3c.org/TR/REC-xml.

Metadata Transformation and Management with Oracle interMedia

Marco Carrer, Ashok Joshi, Paul Lin, and Alok Srivastava
{*mcarrer,ajoshi,pilin,alsrivas*}@*us.oracle.com*
Oracle Corporation
One Oracle Drive
Nashua, NH 03062

Abstract

The management of multimedia data in object-relational database systems presents a challenging problem for extracting, processing, and managing associated metadata. This information is usually embedded within the media source using proprietary formats, thus not easily accessible in a uniform fashion. This poses a need for a series of structural transformations to ensure that the metadata gathering process produces a unified representation across a multitude of media sources. The ultimate goal of these transformations is to consolidate management efforts for diverse sources of media.

This paper presents the Oracle MediaAnnotator, an extensible architecture developed to be used with Oracle8i interMedia. The MediaAnnotator supports automatic extraction and transformation of metadata into logical annotations. This approach allows for the creation of unified metadata repositories, which can then be used for indexing and searching. The extensible nature of this architecture makes it applicable to any multimedia domain. The MediaAnnotator leverages Oracle8i interMedia to tie together the multimedia data and its logical annotations; this offers greater manageability of media archives and opens the possibility for new applications which integrate multimedia content with other user data.

1 Introduction

Digital multimedia yields a format that is very different from alphanumeric data. While textual information supports the concepts of alphabetical ordering, indexing, and searching, media data does not. Multimedia formats are designed to fulfill playback rather than manageability requirements. In order to add manageability, it is imperative that metadata be associated with the media [2, 3, 4, 5, 6, 9]. The approaches presented in the literature support the idea that, in order to achieve fast data retrieval through queries, media information must be extracted – automatically or manually – from the raw media data and stored in a different, more readily usable format, which will constitute the input for the query engines. This transformation process is known as *media annotation*. The transformation into annotations – which involves a special processing of a set of inputs to provide an enriched set of contents – allows the employment of conventional database systems to manage digital multimedia.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

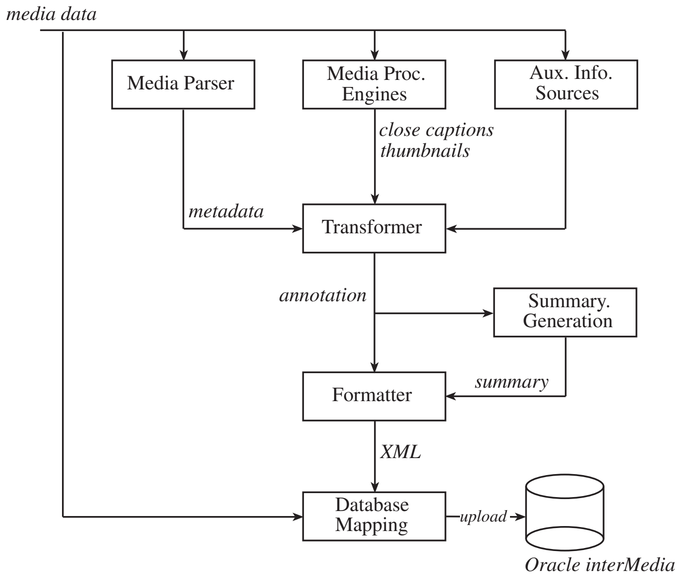

Figure 1: The MediaAnnotator Architecture

The development of Oracle MediaAnnotator has been motivated by the abundance of information already embedded into media sources. This consideration is validated by popular media formats such as QuickTime, from Apple Computer [1], and Advanced Streaming Format (ASF), from Microsoft Corp. [7], both of which are designed to allow for capturing user's as well as system annotations within the format. This information is generally added during media creation and editing. In addition, emerging standards such as Digital Versatile Disc (DVD) and MPEG-7 support even richer metadata, which facilitates the design of automated media management solutions. However, there is no unified way of capturing and using this metadata in applications. The MediaAnnotator addresses this problem by capturing proprietary metadata and transforming it into logical annotations. Its architecture also enables the capturing of associated data which are not available within the media.

Section 2 covers the terminology that will be used throughout this paper. Section 3 offers a detailed discussion of the Oracle MediaAnnotator and, finally, Section 4 concludes the paper.

2 Terminology

For simplicity, we define the following terminology used throughout this paper: raw digital video and audio data are collectively referred to as media data. Descriptions for media data will be referred to as metadata. Metadata and annotations are used interchangeably in this paper identifying a collection of easily indexable attributes (media title, copyright, etc.) or properties.

3 Oracle MediaAnnotator

3.1 Architecture Overview

Oracle MediaAnnotator addresses the need to manage, index, and search digital multimedia by capturing and organizing the associated metadata, which semantically describe media content. This information is captured and re-structured into a searchable form which is suitable for management with Oracle *inter*Media [8].

The MediaAnnotator provides a framework to capture, transform, process, and store metadata. This framework is extensible to allow for user-defined components to be plugged in at every step. Fig. 1 gives a graphical overview of the framework. As shown in the diagram, media data flows through the system along multiple paths. The parser takes media data to extract metadata while the media processing engine generates information based upon media contents. The generated information is then combined with any auxiliary metadata sources, and transformed into annotations. At this stage a summary of the media data can also be generated based on the information captured so far. After being formatted into a searchable representation, annotations and the original media data are uploaded into the database.

The following sections offer detailed descriptions of the components of the system depicted in the Fig. 1.

3.2 Media Parser

The media parser extracts metadata embedded in the digital media according to the file format specifications, which contain instructions on how to parse and extract this information. The MediaAnnotator uses the mimetype of the media source to dynamically load the appropriate parsers for the media. This enables user-defined parsers to be plugged into the MediaAnnotator framework at run-time, thereby extending the range of media formats handled by the system.

3.3 Media Processing Engine

The media processing engine is responsible for generating additional information by analyzing the actual media content. The output of this engine is often time-based; closed-captions, thumbnails, embedded links (URL flipping), and sample clips are some examples. Collectively this information can be viewed as a set of time-based snapshots of the media. Consequently, advanced queries can be performed, producing results with time stamps which can then be used to seek to specific positions within the media.

3.4 Auxiliary Metadata Sources

Auxiliary metadata sources provide the information which is not obtainable by processing the media itself. For example, audio compact discs do not carry any meta-information along with the physical media; it is therefore necessary to gather metadata from auxiliary sources such as user's input, or look-up services on the Internet.

3.5 Transformer

The transformer combines the media information collected thus far to construct unified logical annotations, which contain attribute value pairs as well as time-based samples, describing the media. For example, the logical annotation for an audio compact disc will feature attributes such as the title, artist, duration, and number of tracks as well as audio clips for each track.

The primary task here is to organize the captured attributes and partition them semantically. Each resulting partition constitutes a logical annotation. This structuring of the metadata provides a facilitated method of managing the media. In particular, the effect of such a transformation is an abstraction layer above the diverse multitude of media formats. The client of annotations is shielded from having to understand the format and storage specifications of the original media source. It is now possible to manage the data in a semantically rich manner.

The MediaAnnotator allows users to override the predefined set of annotations or define a completely new set. Similar to parsers, annotations are dynamically loaded based upon the mimetype of the media.

3.6 Summary Generator

Logical annotations can be processed to generate a summary of the media data. The summary generator accomplishes this task according to user's specified guidelines. For example, a song can be summarized by grouping together the performer's name, the song title, and a song clip. The summaries are especially useful for quick browsing of media catalogs.

3.7 Formatter

The formatter is responsible for transforming the logical annotations as well as the summaries into a form which is searchable and manageable by databases. A well-defined XML structure is used by the MediaAnnotator to store this information, hence a unified representation for the metadata is achieved.

3.8 Database Mapper

Database Mapping is the final step of the transformation chain and completes the database population process. During this step, the MediaAnnotator uploads the media and the associated XML document, produced by the formatter, into the Oracle database. Database mapper leverages the media support offered by Oracle *inter*Media, the design of which allows for simultaneous storage of the actual media data and its corresponding metadata. The MediaAnnotator maps the physical properties captured in a logical annotation, to fields of an *inter*Media object. In addition, the XML representation, which includes content attributes, is also stored within the object. As a result, a self-contained repository, for the media data and its description, is created in the Oracle database. This repository can now be indexed with *inter*Media indexing techniques [8], enabling advanced searches on the multimedia data. The subject of indexing techniques is beyond the scope of this paper and is not discussed.

4 Conclusion

In this paper, we have presented an architecture which addresses the problem of managing multimedia data through a set of transformations. The extensibility of the MediaAnnotator makes it applicable to a wide variety of media domains. The MediaAnnotator is extensible in both the understanding of new media formats, and the grouping of attributes into meaningful logical annotations. The media and its logical annotations are stored into a single uniform repository within an Oracle database. This final transformation to a database stored structure, such as an Oracle8*i* *inter*Media object, takes advantage of the built-in textual and object management functionality of an OR-DBMS and completes the process of a semi-automated solution for media asset management.

References

[1] Apple Computer Inc. *QuickTime*. Inside Macintosh. Addison-Wesley Publishing Company, 1993.

[2] K. Böhm and T. C. Rakow. Metadata for Multimedia Documents. *SIGMOD Record*, 23(4):21–26, December 1994.

[3] M. Carrer, L. Ligresti, G. Ahanger, and T.D.C. Little. An Annotation Engine for Supporting Video Database Population *Multimedia Tools and Applications*, 5(3):233-258, November 1997.

[4] G. Davenport, T. A. Smith, and N. Pincever. Cinematic Primitives for Multimedia. *IEEE Computer Graphics and Applications*, pages 67–74, July 1991.

[5] W. I. Grosky, F. Fotouhi, and I. K. Sethi. Using Metadata for the Intelligent Browsing of Structured Media Objects. *SIGMOD Record*, 23(4):49–56, December 1994.

[6] W. Klaus and A. Sheth. Metadata for Digital Media: Intruduction to the Special Issue. *SIGMOD Record*, 23(4):19–20, December 1994.

[7] Microsoft Corporation *Advanced Streaming Format (ASF) Specification*. Public Specification Version 1.0, 1998.

[8] Oracle Coporation *Oracle8i interMedia Audio, Image, and Video User's Guide and Reference* Release 8.1.5 (A67299-01), 1999.

[9] Y. Tonomura, A. Akutsu, Y. Taniguchi, and G. Suzuki. Structured Video Computing. *IEEE Multimedia*, pages 34–43, Fall 1994.

Flexible Database Transformations: The SERF Approach *

Kajal T. Claypool and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{kajal|rundenst}@cs.wpi.edu

Abstract

Database transformation is a critical task that occurs in many different domains. Schema evolution is one important class of problems for database transformations. In our work, we use existing technology and standards (ODMG, OQL, basic schema evolution primitives) to bring flexibility, extensibility and re-usability to current schema evolution systems, thus allowing the users to conveniently specify any customized transformation of their choice. We also investigate the re-usabilty of our framework to other applications beyond schema evolution such as web re-structuring.

Keywords: Schema Evolution, Transformation Templates, Object-Oriented Databases,Modeling Database Dynamics, OQL, ODMG, Schema Consistency.

1 Introduction

The age of information management and with it the advent of increasingly sophisticated technologies have kindled a need in the database community and others to *transform* existing systems and move forward to make use of these new technologies. Legacy application systems are being transformed to newer state-of-the-art systems, information sources are being mapped from one data model to another, a diversity of data sources are being transformed to load, cleanse and consolidate data into modern data-warehouses.

One important class of data transformations are schema evolution tools that do on-line transformation of database systems by modifying both the schema as well as the underlying data objects without bringing the system down [Zic92]. For this, most object-oriented database systems (OODB) today support a *pre-defined* taxonomy of *simple fixed-semantic* schema evolution operations [BKKK87, Tec94, BMO$^+$89, Inc93, Obj93]. More advanced changes such as combining two types have also recently been looked at by Breche [Bré96] and Lerner [Ler96], but are still limited to being a *fixed* set. Anything beyond the *fixed* taxonomy often requires application users to

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was supported in part by the NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and Informix for software contribution. Special thanks also goes to the PSE Team specifically, Gordon Landis, Sam Haradhvala, Pat O'Brien and Breman Thuraising at Object Design Inc. for not only software contributions but also for providing us with a customized patch of the PSE Pro2.0 system that exposed schema-related APIs needed to develop our tool.

write ad-hoc programs to accomplish such transformations. Such programs are very specific and in general cannot be shared across applications and since there is no system-level support for maintaining the consistency of the system, they are more prone to errors. To address these limitations of the *current* transformation technology, we have proposed the SERF framework which aims at providing a rich environment for doing complex user-defined transformations *flexibly*, *easily* and *correctly* [CJR98b]. In this paper we give an overview of the SERF framework, its current status and the enhancements that are planned for the future. We also present an example of the application of SERF to a domain other than schema evolution, i.e., the web re-structuring.

The rest of the paper is organized as follows. Section 2 gives an overview of the key concepts of the SERF Framework. Section 3 discusses some of the more advanced features which are now being added on to SERF to increase the usability and dependability of the SERF system. Section 3.3 outlines the application of SERF to other domains. We conclude in Section 4.

2 The Basic SERF Framework

The SERF framework addresses the limitation of current OODB technology that restricts schema evolution to a *predefined* set of simple schema evolution operations with *fixed* semantics [BKKK87, Tec94, BMO+89, Inc93, Obj93]. With the SERF framework we can offer *arbitrary user-customized* and possibly *very complex* schema evolution operations such as *merge*, *inline* and *split* [Ler96, Bré96] without users having to resort to writing ad-hoc code. Moreover, for each transformation type there can be many different semantics based on user preferences and application needs. For example two classes can be merged by doing a union, an intersection or a difference of their attributes. Similarly the deletion of a class that has a super-class and several sub-classes can be done by either propagating the delete of the inherited attributes through all sub-classes, or by moving the attributes up to the super-class, or by moving them down to all the sub-classes, or any composition of the above.

Our approach is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives, where each basic primitive is an invariant-preserving atomic operation with fixed semantics provided by the underlying OODB system. In order to effectively combine these primitives and to be able to perform arbitrary transformations on objects within a complex transformation, we rely on a standard query language namely OQL [Cat96]. The alternative approach to define a new language for specifying the database transformations has been explored in the literature [DK97] (also see this issue). In our work, we demonstrate that a language such as OQL is sufficient for accomplishing schema evolution.

We illustrate the steps involved in a schema evolution transformation using the example of *Inlining* which is defined as the replacement of a referenced type with its type definition [Ler96]. For example in Figure 1 the `Address` type is inlined into the `Person` class, i.e., all attributes defined for the `Address` type (the referenced type) are now added to the `Person` type resulting in a more complex `Person` type. Figure 2 shows the *Inline* transformation expressed in our framework using OQL, schema modification primitives such as *add_attribute()*, and system-defined update methods such as *obj.set()*.

In general in a SERF transformation there are three types of steps:

- **Step A: Change the Schema.** We require that all structural changes, i.e., changes to the schema, are exclusively made through the schema evolution primitives. This helps us guarantee schema consistency after the application of a transformation [CJR98b]. For example, **Step A** in Figure 2 shows the addition of the attributes `Street, City` and `State` via the *add_attribute* schema evolution (SE) primitive to the `Person` class.

- **Step B: Query the Objects.** Prior to performing object transformations, we need to must obtain the handle for objects involved in the transformation process. This may be objects from which we copy object values (e.g., `Address` objects in **Step B**), or objects that get modified themselves (e.g., `Person` objects in **Step C**).

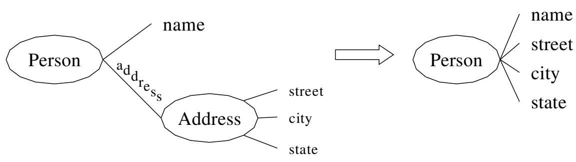

Figure 1: Example of an Inline Transformation.

```
// Add the required attributes to the Person class
add_attribute (Person, Street, String," ");
add_attribute (Person, City, String," ");                     Step A
add_attribute (Person, State, String," ");

// Get all the objects for the Person class
define extents() as                                           Step B
   select c
   from Person c;

// Update all the objects
for all obj in extents():
   obj.set (obj.Street, valueOf(obj.address.Street)) AND      Step C
   obj.set (obj.City, valueOf(obj.address.City))     AND
   obj.set (obj.State, valueOf(obj.address.State)

// Delete the address attribute                               Step B
delete_attribute (Person, address);
```

Figure 2: Inline Transformation Expressed in OQL with Embedded Evolution Primitives.

- **Step C: Change the Objects.** The next step to any transformation logically is the transformation of the objects to conform to the new schema. Through **Step B**, we already have a handle to the affected object set. **Step C** in Figure 2 shows how a query language like OQL and system-defined update methods, like *obj.set*(...), can be used to perform object transformations.

The transformation uses the query language to invoke the schema evolution primitives for schema changes and the system-defined functions for object updates, as in **Steps A** and **C**. Thus we require the capability to invoke method calls as part of a query specification, which is indeed supported by OQL [Cat96].

SERF Template. A SERF transformation as given in Figure 2 *flexibly* allows a user to define different semantics for any type of schema transformation. However, these transformations are *not re-usable* across different classes or different schemas. For example, the *inline* transformation shown in Figure 2 is valid only for the classes `Person` and `Address`. To address this, we have introduced the notion of templates in the SERF framework [CJR98b]. A template uses the query language's ability to query over the meta data (as stated in the ODMG Standard) and is enhanced by a name and a set of parameters to make transformations *generic* and *re-usable*. Figure 3 shows a templated form of the transformation presented in Figure 2. The section of the template marked **Step D** shows the steps required to achieve the effect of **Step A** in Figure 2 in a general form. Thus when this inline template shown in Figure 3 is instantiated with the variables `Person` and `address` it results in the SERF transformation in Figure 2. A template is thus an arbitrarily complex transformation that has been encapsulated and generalized via the use of the ODMG Schema Repository, a name and a set of parameters.

SERF Template Library. The SERF templates can be collected into a **template library** which in turn can be grouped in many different ways, for example by domain such as templates for doing data cleansing, or by object model such as templates for the graph or the web model, thus providing a valuable *plug-and-play* resource to the transformation community. Our overall goal is thus to provide SERF as a value-added service layer on top of existing database systems as the template library can ideally be plugged in for any SERF system.

```
begin template inline (className, refAttrName)
{

  refClass = element (
               select a.attrType
               from MetaAttribute a
               where a.attrName = $refAttrName
               and   a.classDefinedIn = $className; )

   define localAttrs(cName) as
      select c.localAttrList
      from MetaClass c
      where c.metaClassName = cName;

   // get all attributes in refAttrName and add to className
   for all attrs in localAttrs(refClass)
      add_atomic_attribute ($className, attrs.attrName,
                             attrs.attrType, attrs.attrValue);

   // get all the extent
   define extents(cName) as
      select c
      from cName c;

   // set: className.Attr = className.refAttrName.Attr
   for all obj in extents($className):
      for all Attr in localAttrs(refClass)
         obj.set (obj.Attr, valueOf(obj.refAttrName.Attr))

   delete_attribute ($className, $refAttrName);
}

end template

Legend: cName: OQL variables
        $className: template variables
        refClass    user variables
```

```
begin template inline (className, refAttrName)
{
  REQUIRES:

    className must exist in the currentSchema AND
    refAttrName must exist in className AND
    domain(refAttrName) must exist in currentSchema AND
    className != domain(refAttrName)


        Body of Inline Template


  ENSURES:
    className exists in thisSchema AND
    refAttrName does not exist in thisSchema AND
    Properties(domain(refAttrName)) is subset of className

}
end template
```

Figure 3: Left: The Inline Template; Right: The Contractual Form of the Inline Template.

SERF System - OQL-SERF. An implementation of SERF, OQL-SERF, is currently being developed at Worcester Polytechnic Institute. It is based on the ODMG standard and uses the ODMG object model, the ODMG Schema Repository definition, as well as OQL. The system is being implemented entirely in Java and uses Object Design's Persistent Storage Engine (PSE) for Java as its back-end database [RCL+99].

3 Beyond the Base SERF System

The goal of the SERF project is now to increase usability, utility and applicability of the SERF framework to transformation problems beyond OODB evolution. We have started work on providing an assurance of consistency for the users and a semantic optimizer to improve the performance of the transformations, and have also started looking at domains beyond schema evolution for the applicability of this transformation framework.

3.1 Consistency Management

Consistency management is the definition of consistency violations, re-establishment of consistency following violations, and the meaningful manipulation of objects that are not in a consistent state. This is a key problem for complex applications in general and in the face of database transformations it is an even more critical problem. One example for the use of consistency violation detection in the SERF environment where we are dealing with an expensive transformation process is the early (prior to execution) detection of erroneous templates via the consistency manager. This would help improve performance by saving the cost of rollback.

For this purpose, we have developed a model that allows for the specification of consistency constraints for the SERF templates using the *contract* model [Mey92]. Based on this model, we are developing a consistency checker that allows us to detect not only the violation of the consistency constraints but also helps in the verification of the templates. Figure **??** shows the *Inline* template written with *contracts* in easily understandable *English*. We are in the process of developing a language for the specification of *contracts*. Within a SERF template *contracts* serve both as a vehicle for a declarative specification of the behavior of a template as well as for the specification of the constraints under which a SERF template can be applied to the underlying system. Thus, beyond the advantage of violation detection, the *contracts* give us the added advantage to now provide a more sophisticated search mechanism for templates in our libraries based on their declarative behavioral descriptions.

3.2 Semantic Optimizer

Database transformation is an extremely expensive process both in terms of time as well as system resources. A simple schema evolution operation such as *add_attribute* for a large number of objects (approx. 100,000 objects) can take on the order of hours for processing. Hence a complex operation as specified by a SERF transformation can take even longer. We thus have developed the CHOP optimizer that reduces the number of operations in a sequence of schema evolution operations and have shown it to have significant savings [CNR99]. However, since SERF templates may inter-leave OQL queries with schema evolution operations, our current CHOP techniques alone are not sufficient for their optimization. Thus, we are in the process of developing query re-writing techniques with emphasis on reducing the number of expensive method calls (in our case schema evolution primitives) in a SERF template by exploiting existing CHOP techniques. Beyond the evolution domain the time savings by these optimizations may potentially be of an even bigger advantage, for example, in doing transformations for data integration over several legacy systems,

3.3 Application of SERF to Other Problem Domains - Re-WEB

The SERF framework is directly applicable to many other domains that are volatile by nature and thus have an extensive need for re-structuring the underlying structure. In particular, the SERF system can be used for doing transformations above and beyond the *fixed* basic primitives that are provided by current systems. As an example, we have already applied the SERF framework as a tool for re-structuring web sites (this is part of the **Re-WEB tool** [CRCK98] which generates and re-structures web sites.). For this purpose, we have developed a web mapping for the direct generation of web-sites from the schema of an ODMG based database. The key of the Re-WEB tool is the application of the SERF technology to transform the underlying database to produce a diversity of views that match the desired web layout. A template library of frequent web-transformations is a distinct advantage of ReWEB to achieve for example personalized web pages in a fast and efficient manner.

This is but one of many possible applications of SERF. While some small extensions might be required for the SERF system, the key concepts are applicable to many key areas such as for creating data warehouses, for data cleansing, and for addressing schema integration problems.

4 Conclusion

The SERF framework brings to the user a general-purpose transformation framework with the advantages that have existed within some programming language environments, such as templates, libraries, consistency management, etc., but have been slow to propagate to the database arena. The SERF framework gives the users the *flexibility* to define the re-structuring semantics of their choice; the *extensibility* of defining new complex re-structuring transformations meeting specific requirements; the *generalization* of these transformations through the notion of templates; the *re-usability* of a template from within another template; the *ease* of template specification by programmers and non-programmers alike; the *soundness* of the transformations in terms of assuring schema consistency; and the *portability* of these transformations across OODBs as libraries.

An ODMG based implementation of the SERF framework, OQL-SERF [CJR98a], is currently underway at the Worcester Polytechnic Institute and the system is also being demonstrated at SIGMOD'99 [RCL+99].

Acknowledgments. The authors would like to thank students at the Database Systems Research Group (DSRG) at WPI for their interactions and feedback on this research. In particular, we would like to thank Jing Jin and Chandrakant Natarajan for their initial work on SERF. We would also like to thank Anuja Gokhale, Parag Mahalley, Swathi Subramanian, Jayesh Govindrajan, Stacia De Lima, Stacia Weiner, Xin Zhang and Ming Li for their help with the implementation of OQL-SERF.

References

References

[BKKK87] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.

[BMO+89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.

[Bré96] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

[Cea97] R.G.G. Cattell and et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

[CJR98a] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.

[CJR98b] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.

[CNR99] K.T. Claypool, C. Natarajan, and E.A. Rundensteiner. CHOP: An Optimizer for Schema Evolution Sequences. Technical Report WPI-CS-TR-99-06, Worcester Polytechnic Institute, February 1999.

[CRCK98] K.T. Claypool, E. A. Rundensteiner, L. Chen, and B. Kothari. Re-usable ODMG-based Templates for Web View Generation and Restructuring. In *CIKM'98 Workshop on Web Information and Data Management (WIDM'98), Washington, D.C., Nov.6*, 1998.

[DK97] S.B. Davidson and A.S. Kosky. WOL: A Language for Database Transformations and Constraints. In *IEEE Int. Conf. on Data Engineering*, pages 55–65, 1997.

[Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

[Ler96] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, Computer Science Department, 1996.

[Mey92] B. Meyer. Applying "Design By Contract". *IEEE Computer*, 25(10):20–32, 1992.

[Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems*. Object Design Inc., December 1993.

[RCL+99] E.A. Rundensteiner, K.T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD'99*, 1999.

[Tec94] O_2 Technology. *O_2 Reference Manual, Version 4.5, Release November 1994*. O_2 Technology, Versailles, France, November 1994.

[Zic92] R. Zicari. A Framework for O_2 Schema Updates. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Pub., 1992.

Specifying Database Transformations in WOL *

Susan B. Davidson
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
Email: susan@central.cis.upenn.edu

Anthony S. Kosky
Gene Logic Inc.
Bioinformatics Systems Division
2001 Center Str, Suite 600
Berkeley, CA 94704
Email: anthony@genelogic.com

Abstract

WOL is a Horn-clause language for specifying transformations involving complex types and recursive data-structures. Its declarative syntax makes programs easy to modify in response to schema evolution; the ability to specify partial clauses facilitates transformations when schemas are very large and data is drawn from multiple sources; and the inclusion of constraints enables a number of optimizations when completing and implementing transformations.

1 Introduction

Database transformations arise in a number of applications, such as reimplementing legacy systems, adapting databases to reflect schema evolution, integrating heterogeneous databases, and mapping between interfaces and the underlying database. In all such applications, the problem is one of mapping instances of one or more *source* database schemas to an instance of some *target* schema.

The problem is particularly acute within biomedical databases, where schema evolution is pushed by rapid changes in experimental techniques, and new, domain specific, highly inter-related databases are arising at an alarming rate. A listing of the current major biomedical databases indicates that very few of these databases use commercial database management systems (see `http://www.infobiogen.fr/services/dbcat/`). One reason for this is that the data is complex and not easy to represent in a relational model; the typical structures used within these systems include sets, deeply nested record structures, and variants. A transformation language for this environment must therefore be easy to modify as the underlying source database schemas evolve, and capture the complex types used. Mappings expressed in the language must also explicitly resolve incompatibilities between the sources and target at all levels – in the choice of data-model and DBMS, the representation of data within a model, and the values of instances.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, ARPA N00014-94-1-1086 and DOE DE-AC03-76SF00098.

As an example of a transformation, suppose we wish to integrate a database of US Cities-and-States with another database of European-Cities-and-Countries. Their schemas are shown in Figures 1 (a) and (b) respectively, and use graphical notation inspired by [AH87]. Boxes represent *classes* which are finite sets of objects, and arrows represent *attributes*, or functions on classes.

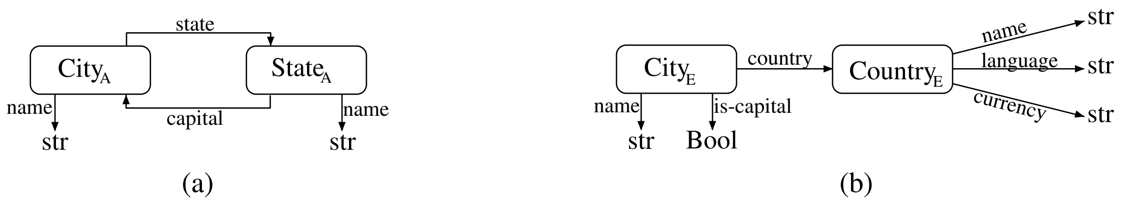

Figure 1: Schemas for US Cities and States (a) and European Cities and Countries (b)

The first schema has two classes: $City_A$ and $State_A$. The $City_A$ class has two attributes: *name*, representing the name of a city, and *state*, which points to the state to which a city belongs. The $State_A$ class also has two attributes, representing its name and its capital city. The second schema also has two classes, $City_E$ and $Country_E$. The $City_E$ class has attributes representing its name and its country, but in addition has a Boolean-valued attribute *is_capital* which represents whether or not it is the capital city of a country. The $Country_E$ class has attributes representing its name, currency and the language spoken.

A schema representing one possible integration of these two databases is shown in Figure 2, where the "plus" node indicates a variant. Note that the *City* classes from both source databases are mapped to a single class $City_T$ in the target database. The *state* and *country* attributes of the *City* classes are mapped to a single attribute *place* which takes a value that is either a *State* or a *Country*. A more difficult mapping is between the representations of capital cities of European countries. Instead of representing whether a city is a capital or not by means of a Boolean attribute, the *Country* class in our target database has an attribute *capital* which points to the capital city of a country. To resolve this difference in representation a straightforward embedding of data will not be sufficient. Constraints on the source database, ensuring that each *Country* has exactly one *City* for which the *is_capital* attribute is true, are also necessary in order for the transformation to be well defined.

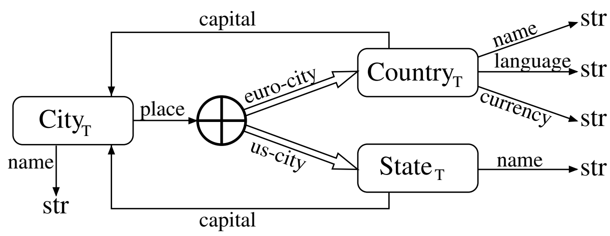

Figure 2: An integrated schema of European and US Cities

To specify exactly how this and other transformations involving complex types and recursive structures is to be performed, we have developed a language called *WOL* (Well-founded Object Logic). A number of considerations have gone into the design of this language. First, a data transformation language differs from a database query language in that entire database instances are potentially being manipulated and created. This implies a careful balance between expressivity and efficiency. Although the transformation language should be sufficiently expressive to specify all ways in which data might relate between one or more source databases and a target database, an implementation of a transformation should be performed in one pass over the source databases. This curtails the inclusion of expensive operations such as transitive closure.

Second, the size, number and complexity of schemas that may be involved in a transformation leads to a need for partiality of rules or statements of a transformation language, and for the ability to reason with constraints. Schemas – especially in biomedical applications, which formed the initial impetus for this work – can be complex, involving many, deeply nested attributes. Values for attributes of an object in a target database may be drawn from many different source database instances. In order to prevent the complexity of transforation rules becoming unmanageable, it is therefore necessary to be able to specify the transformation in a step-wise manner in which individual rules do not completely describe a target object.

Third, constraints can play a part in determining and optimizing transformations; conversely, transformations can imply constraints on their source and target databases. Many of the constraints that arise in transformations fall outside of those supported by most data-models (keys, functional and inclusion dependencies and so on) and may involve multiple databases. It is therefore important that a transformation language be capable of expressing and interacting with a large class of constraints.

In the remainder of this paper, we describe the database transformation language *WOL* and how transformation programs are implemented in a prototype system called *Morphase*[1].

2 Data Model

The data model underlying *WOL* supports object-identities, classes and complex data-structures. Formally, we assume a finite set $\mathcal{C}$ of *classes* ranged over by $C, C', \ldots$, and for each class C a countable set of *object identities* of class C. The *types* over $\mathcal{C}$, ranged over by $\tau, \ldots$, are given by the syntax

$$\tau \quad ::= \quad C \mid \underline{b} \mid (a : \tau, \ldots, a : \tau) \mid \langle\!| a : \tau, \ldots, a : \tau |\!\rangle \mid \{\tau\}$$

Here $\underline{b}$ are the built in *base types*, such as *integer* or *string*. *Class types* C, where $C \in \mathcal{C}$, represent object-identities of class C. $\{\tau\}$ are set types. $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ constructs record types from the types $\tau_1, \ldots, \tau_n$, whereas $\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ builds variant types from the types $\tau_1, \ldots, \tau_n$. A value of a record type $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ is a tuple with k fields labeled by $a_1, \ldots, a_k$, such that the value of the ith field, labeled by a_i, is of type τ_i. A value of a variant type $\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ is a pair consisting of a label a_i, where $1 \leq i \leq k$, and a value of type τ_i.

A database schema can be characterized by its classes and their associated types. For example, the US Cities and States schema has two classes representing cities and states. Each city has a *name* and a *state*, and each state has a *name* and a *capital city*. The set of classes for the schema is therefore $\mathcal{C}_A \equiv \{\mathit{City}_A, \mathit{State}_A\}$ and the associated types are

$$\tau^{\mathit{City}_A} \equiv (\mathit{name} : \mathit{str},\ \mathit{state} : \mathit{State}_A),\quad \tau^{\mathit{State}_A} \equiv (\mathit{name} : \mathit{str},\ \mathit{capital} : \mathit{City}_A)$$

The European Cities and Countries schema has classes $\mathcal{C}_E \equiv \{\mathit{City}_E, \mathit{Country}_E\}$ and associated types

$$\begin{aligned} \tau^{\mathit{City}_E} &\equiv (\mathit{name} : \mathit{str},\ \mathit{is\_capital} : \mathit{Bool}, \mathit{country} : \mathit{Country}_E) \\ \tau^{\mathit{Country}_E} &\equiv (\mathit{name} : \mathit{str},\ \mathit{language} : \mathit{str}, \mathit{currency} : \mathit{str}) \end{aligned}$$

3 The *WOL* Language

WOL is a Horn-clause language designed to deal with the complex recursive types definable in the model. The specification of a transformation written in *WOL* consists of a finite set of *clauses*, which are logical statements describing either constraints on the databases being transformed, or part of the relationship between objects in

[1] *Morphase* has no relation to the god of slumber, Morpheus, rather it is an enzyme (-ase) for morphing data.

the source databases and objects in the target database. Each clause has the form $head \Longleftarrow body$ where $head$ and $body$ are both finite sets of *atomic formulae* or *atoms*.

The meaning of a clause is that, if all the atoms in the body are true, then the atoms in the head are also true. More precisely, a clause is *satisfied* iff, for any instantiation of the variables in the body of the clause which makes all the body atoms true, there is an instantiation of any additional variables in the head of the clause which makes all the head atoms true.

For example, to express the constraint that the capital city of a state must be in that state one would write

$$X.state = Y \Longleftarrow Y \in State_A, X = Y.capital; \qquad (C1)$$

This clause says that for any object Y occurring in the class $State_A$, if X is the $capital$ city of Y then Y is the $state$ of X. Here the body atoms are $Y \in State_A$ and $X = Y.capital$, and the head atom is $X.state = Y$. Each atom is a basic logical statement, for example saying that two expressions are equal or one expression occurs within another.

Constraints can also be used to define keys. In our database of Cities, States and Countries, we would like to say that a Country is uniquely determined by its $name$, while a City can be uniquely identified by its $name$ and its $country$. This can be expressed by the clauses

$$X = Mk^{City_T}(name = N,\ country = C) \Longleftarrow\ X \in City_T,\ N = X.name,\ C = X.country; \qquad (C2)$$
$$Y = Mk^{Country_T}(N) \Longleftarrow\ Y \in Country_T,\ N = Y.name; \qquad (C3)$$

Mk^{City_E} and $Mk^{Country_E}$ are examples of *Skolem functions*, which create new object identities associated uniquely with their arguments. In this case, the $name$ of a City and the $country$ object identity are used to create an object identity for the City.

In addition to expressing constraints about individual databases, WOL clauses can be used to express *transformation clauses* which state how an object or part of an object in the target database arises from various objects in the source and target databases. Consider the following clause

$$\begin{aligned}&X \in Country_T,\ X.name = E.name, && (T1)\\ &X.language = E.language,\ X.currency = E.currency \Longleftarrow\ E \in Country_E;\end{aligned}$$

This states that for every $Country$ in our European Cities and Countries database there is a corresponding $Country$ in our target international database with the same name, language and currency.

A similar clause can be used to describe the relationship between European $City$ and $City$ in our target database:

$$\begin{aligned}Y \in City_T,\ Y.name = E.name, Y.place = ins_{\text{euro-city}}(X) \Longleftarrow\ & E \in City_E,\ X \in Country_T, \qquad (T2)\\ & X.name = E.country.name;\end{aligned}$$

Note that the body of this clause refers to objects both in the source and the target databases: it says that if there is a City, E, in the European Cities database and a Country, X, in the target database with the same $name$ as the $name$ of the $country$ of E, then there is a City, Y, in the target database with the same $name$ as E and with $country$ X. ($ins_{\text{euro-city}}$ accesses the $euro\_city$ choice of the variant).

A final clause is needed to show how to instantiate the $capital$ attribute of $City$ in our target database:

$$\begin{aligned}X.capital = Y \Longleftarrow\ & X \in Country_T,\ Y \in City_T, Y.place = ins_{\text{euro-city}}(X),\ E \in City_E, \qquad (T3)\\ & E.name = Y.name,\ E.state.name = X.name, E.is\_capital = True;\end{aligned}$$

Notice that the definition of $Country$ in our target database is spread over multiple WOL clauses: clause (T1) describes a country's $name$, $language$ and $currency$ attributes, while clause (T3) describes its $capital$ attribute. This is an important feature of WOL, and one of the main ways it differs from other Horn-clause logic based query languages such as Datalog or ILOG[HY90] which require each clause to completely specify a target value. It is possible to combine clauses (T1), (T3) and (C3) in a single clause which completely describes how a Country object in the target database arises. However, when many attributes or complex data structures are involved, or a target object is derived from several source objects, such clauses become very complex and difficult to understand. Further if variants or optional attributes are involved, the number of clauses required may be exponential in the number of variants involved. Consequently, while conventional logic-based languages might be adequate for expressing queries resulting in simple data structures, in order to write transformations involving complex data structures with many attributes, particularly those involving variants or optional attributes, it is necessary to be able to split up the specification of the transformation into small parts involving partial information about data structures.

4 Implementing *WOL* Programs

Implementing a transformation directly using clauses such as (T1), (T2) and (T3) would be inefficient: to infer the structure of a single object we would have to apply multiple clauses. For example clauses (T1), (T3) and (C3) would be needed to generate a single *Country* object. Further, since some of the transformation clauses, such as (T1) and (T3), involve target classes and objects in their bodies, we would have to apply the clauses *recursively*: having inserted a new object into $\mathit{Country}_T$ we would have to test whether clause (T2) could be applied to that *Country* in order to introduce a new City_T object.

Since *WOL* programs are intended to transform entire databases and may be applied many times, we trade off compile-time expense for run-time efficiency. Our implementation therefore finds, at compile time, an equivalent, more efficient transformation program in which all clauses are in *normal form*. A transformation clause in normal form completely defines an insert into the target database in terms of the source database only. That is, a normal form clause will contain no target classes in its body, and will completely and unambiguously determine some object of the target database in its head. A transformation program in which all the transformation clauses are in normal form can easily be implemented in a single pass using some suitable database programming language. In our prototype implementation *Morphase*, the Collection Programming Language (CPL) [BDH+95, Won94] is used to perform the transformations.

Unfortunately, not all complete transformation programs have equivalent normal form transformation programs. Further it is not decidable whether a transformation program is *complete*, that is whether it completely describes how to derive all objects in the target database from the source databases, or whether such an equivalent normal form transformation program exists. Consequently *Morphase* imposes certain syntactic restrictions on transformation programs to ensure that they are *non-recursive*, which are easy to verify computationally and are satisfied by most natural transformations.

Within *Morphase*, constraints play an important role in completing as well as implementing transformations. Constraints on the target database can be used to complete transformations. As an example, *Morphase* will combine clauses (T1) and (T3) with the key constraint on $\mathit{Country}_T$ (C3) to generate a single clause which completely specifies how objects of class $\mathit{Country}_T$ are generated from objects in the US Cities and States database.

In a similar way, constraints on the source databases can play an important part in optimizing a transformation. As an example, suppose the following rule was generated as a result of combining several incomplete clauses:

$$\begin{aligned} & X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{language} = L,\ X.\mathit{currency} = C \Longleftarrow \\ & \qquad Y \in \mathit{Country}_E,\ Y.\mathit{name} = N,\ Y.\mathit{language} = L,\ Z \in \mathit{Country}_E,\ Z.\mathit{name} = N,\ Z.\mathit{currency} = C \end{aligned}$$

Implementing the clause as written would mean taking the product of the source class $\mathit{Country}_E$ with itself, and trying to bind Y and Z to pairs of objects in $\mathit{Country}_E$ which have the same value on their *name* attribute. However if we had a constraint on the source database that specified *name* as a key for $\mathit{Country}_E$ the clause could be simplified to the following, more efficient, form

$$X = Mk^{Country_T}(N), X.language = L, X.currency = C \Longleftarrow Y \in Country_E, Y.name = N, \\ Y.language = L, Y.currency = C$$

5 Conclusions

Data transformation and integration in the context of biomedical databases has been a focus of research at Penn over the past six years. Two languages and systems have resulted: CPL and *WOL*. CPL has primarily been used for querying multiple heterogeneous databases, and has proven extremely effective; it is also used to implement language of *WOL* transformations. While CPL – or OQL, or any other database query language with a sufficiently rich data model – could be used for specifying transformations, they lack several useful features that are present in *WOL*. The first is a declarative syntax, which can be easily modified in response to schema evolution. The second is the ability to specify partial clauses, which we have found extremely useful when many variants are involved (as is the case with ACeDB [TMD92]). The third is the ability to capture and reason about database constraints. Reasoning about constraints is critical when partial clauses are used, since target constraints are used to normalize transformation programs. Complete details on *WOL* and Morphase can be found in [KDB95, Kos96, DK97].

The *WOL* language has also been used independently by researchers in the VODAK project at Darmstadt, Germany, in order to build a data-warehouse of protein and protein-ligand data for use in drug design. This project involved transforming data from a variety of public molecular biology databases, including SWISSPROT and PDB, and storing it in an object-oriented database, ReLiBase. *WOL* was used to specify structural transformations of data, and to guide the implementations of these transformations.

References

[AH87] Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[BDH+95] Peter Buneman, Susan Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A data transformation system for biological data sources. In *Proceedings of 21st International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, August 1995.

[DK97] S.B. Davidson and A. Kosky. WOL: A language for database transformations and constraints. In *Proceedings of the International Conference of Data Engineering*, April 1997.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 455–468, 1990.

[KDB95] A. Kosky, S. Davidson, and P. Buneman. Semantics of database transformations. Technical Report MS-CIS-95-25, Department of Computer and Information Science, University of Pennsylvanic, Philadelphia, PA 19104-6389, July 1995. To appear in Semantics of Databases, edited by L. Libkin and B. Thalheim.

[Kos96] Anthony Kosky. *Transforming Databases with Recursive Data STructures*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, 1996. Available as UPenn Technical Report MS-CIS-96-18.

[TMD92] Jean Thierry-Mieg and Richard Durbin. ACeDB — A C. elegans Database: Syntactic definitions for the ACeDB data base manager, 1992.

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

Transforming Heterogeneous Data with Database Middleware: Beyond Integration

L. M. Haas R. J. Miller B. Niswonger M. Tork Roth P. M. Schwarz E. L. Wimmers
{*laura, niswongr, torkroth, schwarz, wimmers*}*@almaden.ibm.com; miller@cs.toronto.edu*

1 Introduction

Many applications today need information from diverse data sources, in which related data may be represented quite differently. In one common scenario, a DBA wants to add data from a new source to an existing warehouse. The data in the new source may not match the existing warehouse schema. The new data may also be partially redundant with that in the existing warehouse, or formatted differently. Other applications may need to integrate data more dynamically, in response to user queries. Even applications using data from a single source often want to present it in a form other than that it is stored in. For example, a user may want to publish some information using a particular XML DTD, though the data is not stored in that form.

In each of these scenarios, one or more data sets must be mapped into a single target representation. Needed transformations may include schema transformations (changing the structure of the data) [BLN86, RR98] and data transformation and cleansing (changing the the format and vocabulary of the data and eliminating or at least reducing duplicates and errors) [Val, ETI, ME97, HS95]. In each area, there is a broad range of possible transformations, from simple to complex. Schema and data transformation have typically been studied separately. We believe they need to be handled together via a uniform mechanism.

Database middleware systems [PGMW95, TRV96, ACPS96, Bon95] integrate data from multiple sources. To be effective, such systems must provide one or more integrated schemas, and must be able to transform data from different sources to answer queries against these schema. The power of their query engines and their ability to connect to several information sources makes them a natural base for doing more complex transformations as well. In this paper, we look at database middleware systems as tranformation engines, and discuss when and how data is transformed to provide users with the information they need.

2 Architecture of a DB Middleware System

To be a successful data transformation engine for scenarios such as the above, a database middleware system must have several features. Since data these days comes from many diverse systems, it must provide access to a broad range of data sources transparently. It must have sufficient query processing power to handle complex operations, and to compensate for limitations of less sophisticated sources. Some transformation operations (especially the complex ones) require that data from different sources be interrelated in a single query.

We use Garlic [C$^+$95] to illustrate the ideas of this paper. Figure 1 shows Garlic's architecture, which is typical of many database middleware systems [PGMW95, TRV96, ACPS96]. Garlic is primarily a query processor;

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

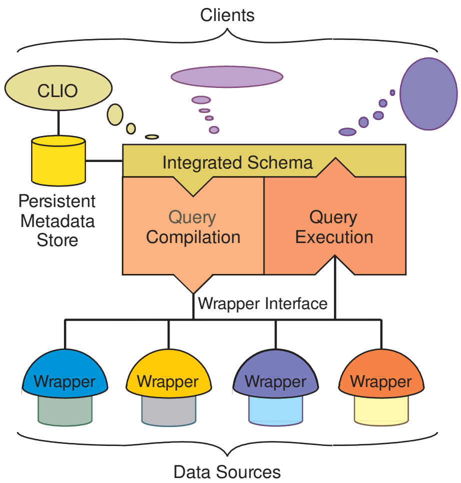

Figure 1: Garlic architecture

it optimizes and executes queries over diverse data sources posed in an object-extended SQL. Garlic's powerful query engine is capable of executing any extended SQL operation against data from any source. In both planning and executing the query, it communicates with *wrappers* for the various data sources involved in the query. Systems of this type have two opportunities to transform data: first, at the wrapper as the data is mapped from the source's model to the middleware model (Section 3), and second, by queries or views against the middleware schema (Sections 4 and 5). However, understanding how the data needs to be transformed is not always simple. The target representation is often only defined implicitly, by existing data. The data integration tool, Clio, shown here and in more detail in Figure 4 will help users understand both their source's data and the target representation and will assist them in creating a mapping between them.

3 Data Transformation at the Wrapper

The most basic tasks of a wrapper are a) to describe the data in its repository and b) provide the mechanisms by which users and the Garlic middleware engine may retrieve that data [RS97]. Since a data source is not likely to conform to Garlic's data model and data format, these two tasks imply that the wrapper must perform some level of schema and data transformation. To make the task of writing a wrapper as easy as possible, the Garlic wrapper architecture tries to minimize the required transformations, but wrappers can do more if desired.

The schemas of individual repositories are merged into the global schema via a wrapper registration step. In this step, wrappers model their data as Garlic objects, and provide an *interface* definition that describes the behavior of these objects. The interface is described using the Garlic Definition Language (GDL), which is a variant of the ODMG Object Description Language [Cat96]. The interface definition provides an opportunity for a wrapper to rename objects and attributes, change types and define relationships even if the data source stores none. For example, a relational wrapper might model foreign keys as relationships. Developing interface files is typically not hard. For simple data sources, it may be best to generate them manually, as simple sources tend to have few object types, usually with fairly simple attributes and behavior. For more sophisticated sources, the process of generating an interface file can often be automated. For example, a relational wrapper can decide on a common mapping between the relational model and the Garlic data model (e.g. tuple = object, column = attribute), and provide a tool that automatically generates the interface file by probing the relational database schema. Wrappers must also provide an *implementation* of the interface which represents a concrete realization of the interface. The implementation cooperates with Garlic to assign a Garlic object id (OID) to its objects, and maps the GDL base types specified in the interface file to the native types of the underlying data source.

Oracle Database Wrapper

Relational Schema

```
CREATE TABLE COUNTRIES {
    NAME VARCHAR(30) NOT NULL,
    CLIMATE VARCHAR(256),
    HIGHESTPEAK NUMBER(4),
    PRIMARY KEY(NAME) }

CREATE TABLE CITIES {
    NAME VARCHAR(40) NOT NULL,
    COUNTRY VARCHAR(30) NOT NULL,
    POPULATION NUMBER(4),
    ELEVATION NUMBER(7,2),
    AREA NUMBER(7,2),
    PRIMARY KEY(NAME),
    FOREIGN KEY(COUNTRY)
        REFERENCES COUNTRIES }
```

Garlic Schema

```
interface Country_Type {
    attribute string name;
    attribute string climate;
    attribute long highest_peak;
};

interface City_Type {
    attribute string name;
    attribute ref¡Country_Type¿ country;
    attribute long population;
    attribute double elevation;
    attribute double area;
};
```

Hotel Web Site Wrapper

```
interface Hotel_Type {
    attribute string name;
    attribute string street;
    attribute string city;
    attribute string country;
    attribute long postal_code;
    attribute string phone;
    attribute string fax;
    attribute short number_of_rooms;
    attribute float avg_room_price;
    attribute short class;
    void display_location();
};
```

Figure 2: Example of wrapper schemas

A hypothetical travel agency application illustrates the kinds of simple schema and data transformations that wrappers can perform. The agency would like to integrate an Oracle database of information on the countries and cities for which it arranges tours with a web site that contains up-to-date booking information for hotels throughout the world. Figure 2 shows the orginal table definitions and the new interface definitions for the two relational tables, and the interface file for the web site. The relational wrapper renamed the `HIGHESTPEAK` field to `highest_peak`, and exposed the foreign key `COUNTRY` on the `CITIES` table as an explicit reference to a `Country` object in the integrated database. The wrapper must be able to map requests for this attribute from the integrated database (in OID format) into the format expected by the relational database (as a string), and vice versa. In addition, the `POPULATION`, `ELEVATION` and `AREA` columns are all stored as type `NUMBER`, yet `population` has type long in the interface file, while `elevation` and `area` are doubles.

Each hotel listing on the web site contains HTML-tagged fields describing that hotel, and a URL to map the location of a particular hotel given its key. In the interface definition file, the HTML fields are represented as attributes of the Hotel object, each with an appropriate data type, though the web site returns all data in string format. The map capability is exposed as the `display-location` method. It is the wrapper's responsibility to map names to the fields on the HTML page, and to convert data from strings into appropriate types.

4 Data Transformation in the Middleware

Views are an important means of reformatting data, especially for middleware, as the data resides in data sources over which the user has little control. Views provide the full power of SQL to do type and unit conversions not anticipated by the wrapper, merging or splitting of attributes, aggregations and other complex functions. In Garlic, *object views* allow further restructuring of data. It is frequently the case that the information about a particular conceptual entity is part of several objects stored in various data sources. However, end-users want to see a single object. An object view creates a new "virtual" object. This virtual object requires no storage since attributes are specified in a query rather than stored as base data. Every virtual object in Garlic is based on another object (which could itself be a virtual object). Garlic uses the OID of the base object as the basis for the virtual object's OID, and provides a function, LIFT, to map the base OID to the virtual object's OID.

One important reason to have virtual objects is to allow new behavior, i.e., new methods, to be defined for these objects. Methods on views can also be used to "lift" methods on the base objects so that virtual objects can retain the base object's functionality. Each method on a virtual object is defined by an SQL query. This query has access to the OID of the virtual object upon which the method is invoked via the keyword *self*, and can find the OID of the base object, if needed. Methods return at most one item; otherwise a run-time error results.

```
interface city_listing_Type {
    attribute string name;
    attribute string country;
    attribute float population_in_millions;
    attribute float elevation_in_meters;
    attribute set¡ref¡Hotel_Type¿¿ hotels;
    string find_best_hotel(IN long budget);
};
```

```
create view city_listing (name, country, population_in_millions, elevation_in_meters, hotels, self)
    as select C.name, C.country, C.population/1000000, C.elevation*0.3048,
        MAKESET(H.OID), LIFT('city_listing', C.OID)
    from Cities C, Hotels H
    where C.name=H.city and UCASE(C.country-¿name)=H.country
    group by C.name, C.country, C.population, C.elevation, C.OID

create method find_best_hotel(long budget)
    return
    select h1.name from unnest self.hotels h1
    where h1.class > all (select h2.rating from unnest self.hotels h2
                          where h2.name ≠ h1.name and h2.avg_room_price ≤ budget)
        and h1.avg_room_price ≤ budget
```

Figure 3: A sample view definition, with method

To register an object view in Garlic, the user must provide both an interface and an implementation (definitions of the view and any methods), as illustrated in Figure 3. This view, based on the `City` objects defined in Section 3, creates `City_Listing` objects that have most of the attributes of a `City` (but omit, for example, area), and an associated set of hotels. The view definition shows how these objects would be created. It uses some of Garlic's object extensions to SQL, including a path expression to get the name of the `City`'s country, and a new aggregate function, `MAKESET`, that creates a set. Note that the LIFT function is used to compute the OID of the new virtual object. All the attributes of the virtual object must be included in the select list. The view definition does some simple unit conversions using arithmetic functions, and uses the uppercase function to map country names from the Oracle database to the names of countries in the Web source. More complex mappings (using translation tables or user-defined functions, for example) would also be possible. The method finds the best hotel within a certain budget in the city on which it is invoked. The budget is an argument of the method. Note the use of *self* to identify the correct set of hotels.

5 Data Mapping and Integration

We have described two components of Garlic that provide important data transformation facilities. The wrappers provide transformations required by the individual data sources, including data model translation and simple schema and data transformations. Object views enhance the wrapper transformations with a general view mechanism for integrating schemas. Object views support integrated cooperative use of different legacy databases, through query language based transformations, such as horizontal or vertical decomposition (or composition) of classes. Such transformations are required to integrate overlapping portions of heterogeneous databases.

In addition, we are working to provide a more powerful suite of schema and data transformations to permit integration of schemas that exhibit schematic discrepancies, and matching of data objects that represent the same real-world entity. Across heterogeneous data sources, different assumptions may have been made as to what data is time invariant and therefore appropriate to include as metadata rather than data. Data under one schema may be represented as metadata (for example, as attribute or class names) in another. Such heterogeneity has been referred to as *schematic heterogeneity*. Traditional query languages are not powerful enough to restructure both data and metadata [Mil98, LSS96]. Likewise, in heterogeneous systems different representations of the same entity in different sources are common. The same object may be identified by a different name in two sources, or even by a different key. Further, different entities may bear similar names in different sources. Identifying equivalent objects and merging them also requires new, powerful transformations [ME97, HS95, Coh98].

In many tasks requiring data translation, the form of the translated data (its schema) is fixed or at least constrained. In a data warehouse, the warehouse schema may be determined by the data analysis and business support tasks the warehouse must support. As new data sources are added to the warehouse, their schemas must be mapped into the warehouse schema, and equivalent objects must be found and converged. Hence, the integrated

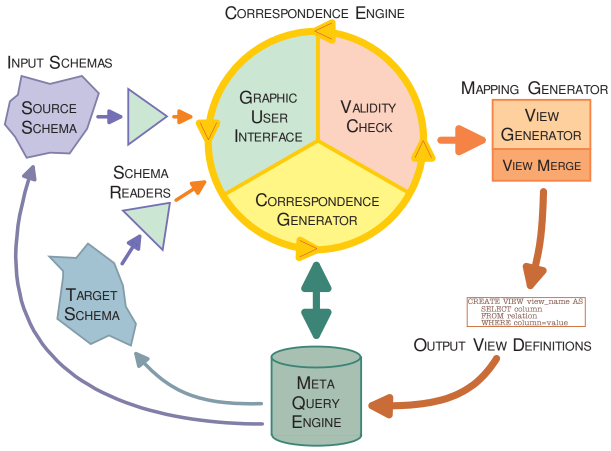

Figure 4: Tool architecture

view is no longer synthesized via a set of transformations local to individual source databases. Rather, it must be possible to discover how a source's schema components and data correspond to a fixed target schema and any existing data. Unlike traditional schema and data integration, this mapping process may require non-local transformations of source schemas and data objects.

As an example, consider the `Hotel_Type` defined in Figure 2. Now imagine that a new source of hotel data has become available, in which some of the data is redundant with the original web source, but most is not. This new source of data has a collection for each country: one for France, one for China, etc. We wish to incorporate this new source in such a way that the user sees only one collection of hotels. However, there are several obstacles to this goal. First, there is schematic heterogenity: in the new source the country name is metadata rather than an attribute value. Second, the system needs to be able to identify when hotels from the two sources are the same. The same hotel may be identified by a different name in the two sources (e.g., "Hyatt St. Claire" vs. "St. Claire Hyatt"), and two different hotels may have the same name (e.g., "Holiday Inn Downtown" exists in many cities). Metadata transformations that use higher order query language operators [Ros92] are needed for the former, dynamic cleansing operations such as joins based on *similarity* [Coh98] for the latter.

6 Building the Integrated Schema

Converting from one data representation to another is time-consuming and labor-intensive, with few tools available to ease the task. We are building a tool, Clio[1], that will create mappings between two data representations semi-automatically (i.e., with user input). Clio moves beyond the state of the art in several ways. First, while most tools deal with either schema integration or data transformation, it tackles both in an integrated fashion. Second, it employs a full database middleware engine, giving it significantly more leverage than the *ad hoc* collections of tools available today, or the lightweight "agents" proposed by others. Third, it will exploit the notion of a target schema, and, where it exists, target data, to make the integration problem more tractable. Fourth, because the middleware engine is being enhanced with more powerful transformation capabilities (Section 5) than most, it will allow more complex transformations of both schema and data. Finally, it will use data mining techniques to help discover and characterize the relationships between source and target schema and data.

The tool has three major components (Figure 4): a set of Schema Readers, which read a schema and translate it into an internal representation (possibly XML); a Correspondence Engine (CE), which finds matching parts of the schemas or databases; and a Mapping Generator, which will generate view definitions to map data in the

[1]Named for the muse of history, so that it will deal well with legacy data!

source schema into data in the target schema. The CE has three major subcomponents: a GUI for graphical display of the schemas and relevant data items, a correspondence generator, and a component to test correspondences for validity. Initially, the CE will expect the user to identify possible correspondences, via the graphical interface, and will provide appropriate data from source and target (using the meta query engine) for verifying the correspondences and identifying the nature of the relationship (again, initially relying on the user). This will be an iterative process. Over time, we anticipate increasing the "intelligence" of the tool using mining techniques so that it can propose correspondences, and eventually, verify them.

Clio will be general, flexible, and extensible. We expect to have a library of code modules (e.g. Java Beans) for transformation operators, which the middleware engine will be able to apply internally. Open research issues include what set of transformations are useful, and whether all transformations (particularly data cleansing) can be done efficiently on the fly. We believe Clio's modular design provides the flexibility required to experiment with a wide range of transformation operators, allowing it to serve as a test bed for further research in this area.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. ACM SIGMOD*, 25(2):137–148, Montreal, Canada, June 1996.

[BLN86] C Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies of database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[Bon95] C. Bontempo. *DataJoiner for AIX*. IBM Corporation, 1995.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[Cat96] R. G. G. Cattell. *The Object Database Standard – ODMG-93*. Morgan-Kaufmann, San Mateo, CA, 1996.

[Coh98] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. ACM SIGMOD*, 27(2):201–212, Seattle, WA, June 1998.

[ETI] ETI - Evolutionary Technologies International. http://www.evtech.com/.

[HS95] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proc. ACM SIGMOD*, 24(2):127–138, San Jose, CA, May 1995.

[LSS96] L. Lakshmanam, F. Sadri, and I. N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[ME97] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. *Proc. of SIGMOD 1997 Workshop on Data Mining and Knowledge Discovery*, May 1997.

[Mil98] R. J. Miller. Using Schematically Heterogeneous Structures. *Proc. ACM SIGMOD*, 27(2):189–200, Seattle, WA, June 1998.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[Ros92] K. A. Ross. Relations with Relation Names as Arguments: Algebra and Calculus. *Proc. ACM PODS*, pages 346–353, San Diego, CA, June 1992.

[RR98] S. Ram and V. Ramesh. Schema integration: Past, present, and future. In A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors, *Management of Heterogeneous and Autonomous Database Systems*. Morgan-Kaufmann, San Mateo, CA, 1998.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. ICDCS*, 1996.

[Val] Vality Technology Incorporated. http://www.vality.com/.

Repository Support for Metadata-based Legacy Migration

Sandra Heiler Wang-Chien Lee Gail Mitchell
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02451
{sheiler,wlee,gmitchell}@gte.com

Abstract

Migrating legacy systems involves replacing (either wholly or partially) existing systems and databases, and complex transformations between old and new data, processes and systems. Correctly performing these activities depends on descriptions of data, and other aspects of the legacy and new systems, and the relationships between them, i.e., metadata. Metadata repositories provide tools for capturing, transforming, storing, and manipulating metadata. They can also store information for managing the migration process itself and for (re)use in other migrations. This paper discusses some of the issues that arise when migrating legacy systems and examines how repository technology can be used to address these issues.

1 Introduction

Modifications to software systems pose a constant challenge to businesses. The complexity of these modifications can range from routine software maintenance (to fix errors, improve performance, or to add or change functionality) to redefinition or replacement of entire business processes and, thus, of the software systems implementing them. Replacing (either wholly or partially) an existing software system is a *legacy migration*: the extant software systems are the legacy; the transformation of data and procedures from the old to new system is a migration process.

Two characteristics of legacy system migration create complexities that do not typically affect other software development:

- The legacy system provides an existing implementation of the target business process. Unlike new development, where the goal is to provide a working implementation at the end, a migration must maintain a working implementation at each step of the way.
- Legacy systems and databases have established relationships with other systems/databases of the enterprise. Data may be input to or output from other systems, or code may be reused in other applications. These relationships must be preserved or migrated during a migration effort.

For example, suppose an organization is migrating its financial systems to a new Enterprise Resource Planning (ERP) implementation (for example, [1]), first replacing the general ledger programs, then replacing the tax

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

accounting system, finally replacing the payroll system. The new general ledger programs might use data produced by the legacy accounts payable programs, and produce data that will be the new source of input for an executive reporting system. Similarly, the legacy HR system and new payroll system might use the same calculations for computing employee contributions to health insurance. At each stage the financial functions in the new system and the remaining functions in the legacy systems must operate properly and interoperate with each other. All the data used and produced by the financial systems must remain correct and consistent regardless of the migration stage and, of course, the systems processes must be correct and consistent at all stages.

Legacy system migration involves not just replacement of existing systems and databases, but complex mappings between old and new data, processes and systems. In most cases, there are no complete replacements during migration and, even once fully migrated, new business process will be implemented with a mix of old and new data and program code. As a result, legacy migration activities depend on descriptions of data, and other aspects of the legacy and new systems, and the relationships between them. This metadata allows designers and engineers to better understand what the systems do, how they do it, and when they do it, as well as what data the systems use or produce and where it comes from [2].

Supporting a legacy migration requires information about data and processes in the old and new systems. In addition, it requires information needed to move from one to the other and to integrate the remaining elements of the old with the elements in the new at each stage of the migration. This information can be represented as metadata describing, for example, components (e.g., usage, functions, interface), data (e.g., source, format, units), code (e.g., source language, change history, versions), processes (e.g., implementations, workflows, requirements), jobs (e.g., control language, performance statistics, hardware requirements), and relationships among the described elements (e.g., instance_of, part_of, computed_by, depends_on_completion_of).

Metadata repositories are designed specifically to store and manipulate these kinds of metadata. Repository technology provides tools for capturing metadata by extracting it from existing systems and transforming it for management in the repository. A repository can also store information about the migration process itself that can be (re)used in other migrations or maintenance, development,etc. projects.

In this paper we examine how repository technology can help with data transformation in the context of legacy system migration. Although we concentrate here on the transformation of data, many of the ideas will also apply to ”transforming” procedures and processes. In the next section we outline the capabilities of available repository products. In Section 3 we present some of the issues related to data transformation when migrating legacy systems, and discuss how repository technology could be put in practice to facilitate migration. We conclude in Section 4, with a discussion of some additional advantages of using a repository during migration.

2 Repository Technology

A metadata repository consists of a set of software tools used for storage, search, retrieval, use and management of metadata. The components of a repository system include a metadatabase and metadata management system, an extensible metamodel and modeling tools, and tools for populating the metadatabase and integrating and accessing the metadata.

The metadatabase and management system. The repository includes a database of metadata and a DBMS to manage it. The repository DBMS provides standard database management facilities, such as persistent storage, data models, keys, certain types of queries, constraints and rules, transactions, data integrity and views, and also provides additional features for dealing with the specific requirements of metadata. Metadata-specific features include built-in complex relationships, navigation, support for long-running transactions, versioning and configuration management. Additional tools may be provided for defining and enforcing metadata security restrictions.

The repository DBMS is a logical system that may be physically instantiated as a single or distributed database. The repository database might be proprietary to a particular repository vendor, or implemented on top of standard database products. Typically a proprietary database will be object-based to support the extensive navigation required to implement and track metadata relationships and versions. A non-proprietary repository might be tightly integrated with a particular database product, or might work with a variety of products via standard interfaces. Requirements for the repository DBMS differ from those for an ordinary DBMS; in particular, the size of a metadatabase is typically orders of magnitude less than what must be supported by a regular DBMS.

Extensible metamodel and modeling tools. A repository system provides a meta-metamodel describing the things the repository needs to know about metadata (e.g., item types, attributes, relationships), and tools for building metamodels to include application-specific metadata descriptions. Generally, a repository product will also provide metamodels describing particular kind of metadata, for example, metamodels for different database schemas, software architecture specifications, or programming languages. Each metamodel provides built-in relationships specific to that type of metadata (e.g., a db2table *has_column* db2column). Metamodels can be modified to adapt to changes in the set of tools or applications supported by the repository, or to the specific characteristics of those tools. A repository product usually provides modeling languages and other tools to facilitate creation and modification of these metamodels.

Tools for populating the metadatabase. The metadata stored in a repository will come from a variety of sources, so a repository provides tools for extracting metadata from these sources (e.g., schemas from databases or data structure specifications from program code). Many software lifecycle management tools automatically store their output in a repository so that metadata generated in one stage of the cycle can be accessed or augmented in other stages of the cycle. For example, most CASE tools use a repository as a central location for capturing the connections between requirements, design and code. Other types of metadata sources do not provide an explicit representation of the metadata, so tools may be provided to extract information from each type of source and generate metadata for the repository[4]. For example, descriptions of a legacy file system might be extracted from COBOL code. Static tools (scanners or parsers) extract metadata from databases, ! ! program code or other sources; dynamic tools (buses) interact with CASE and other software life-cycle support tools to populate the repository as part of the process of using the tool. A specific scanner or bus is needed to obtain metadata from each different type of metadata source.

Tools for integrating metadata. Repositories support sharing metadata among different components or tools by providing the ability to define global models and integrated views of metadata from heterogeneous systems. Views can be defined to allow access to any item in a metadata repository by any application or user (subject to security restrictions) regardless of the originating source's platform, language, application system, etc. For example, a repository could provide a single view of metadata from existing DB2 files on a mainframe database, from old COBOL copybooks, and from a newly designed data model based on a CASE tool run on a UNIX system using C++. In addition, model definitions can be used to specify associations among repository elements that indicate semantic similarities among elements from heterogeneous sources, e.g., that a user-id in an ERWIN-generated data model and a customer_no in an Oracle database schema refer to the same data entities.

Tools for accessing metadata. A repository product also includes tools to facilitate easy access to metadata and customization of the repository. These typically include a user interface (GUI), (often) a Web interface for accessing metadata or Web tools for building such an interface, and APIs and/or a programming language for accessing metadata management functions and building additional, application-specific functionality. In addition, access applications such as impact analysis use the navigational facilities of the repository to provide information about the data relationships. For example, impact analysis tools can be used to estimate the scope of changes in a software maintenance project and determine which modules will be affected by proposed changes.

3 Issues in Data Transformation for Legacy System Migration

Transforming data as part of legacy system migration raises issues that are not necessarily of concern when data transformations are performed as part of other software engineering efforts. We focus here on three issues that typically affect transformation in (or are exacerbated by) migration efforts: data rationalization, interoperability, and process management. We discuss how these issues can be addressed (at least in part) using the tools of metadata repository technology.

Data rationalization. Unlike new software development efforts, legacy system migration starts with data that are part of the legacy systems delivering the specified functionality. An important aspect of migration planning is identifying appropriate sources of needed data and determining what transformations will be required to 1) be compatible with the new implementation, or 2) provide expanded functionality in the new system, and 3) support continuing relationships with other parts of the legacy.

For example, suppose the payroll portion of the migration example of Section 1 moves from a variety of different local systems to a centralized system. This requires identifying appropriate sources for the employee roster, time reporting, deductions, tax data, and so on. These sources may provide initial data for loading the new system, or may be used with wrappers as ongoing sources for the new system. In either case, the rationalization process must determine what kinds of transformations are necessary to align both the syntax and semantics of the legacy data items with the new requirements. Also, if multiple sources are available in the legacy for any part of the needed data, the data rationalization effort must determine which source or sources to transform.

Data rationalization can be supported by an inventory of source data elements, including data formats and meanings, the uses of the data, and other relationships. Such an inventory could be built by loading the repository[1] using data extraction tools that work with the input source types and the repository. It is important to note that the semantics of the data elements is intertwined with the functionality of the software components that create or use the elements. Thus, metadata describing the software components that process the data elements is needed to complete the picture of what data sources are available to support data rationalization.

The *process* of rationalizing data could also be supported using the repository. In particular, the metadata generated during the rationalization process – what decisions were made, why they were made, what items have been mapped, by whom, etc. – could be stored in the repository. This metadata could then be analyzed to determine discrepancies, for example, or referenced to assist in making further decisions about the data items or related items.

Interoperability. In any migration, the new system may receive data feeds from the legacy, may become the source of data for systems that are still part of the legacy, or may be required to trigger updates in the legacy. For example, the new payroll system may get its data feeds from the old time reporting system; the HR system (which is not part of the migration) may depend on the new payroll system to update employee benefits information such as vacation time. Since migration is an incremental process; the new implementation must exchange data with elements of the legacy in each delivered increment of the migrating system (including even the final product, because migration generally preserves parts of the legacy). Thus, support for interoperabililty requires detailed information describing the data elements of the legacy as well as the new system.

These data descriptions require both syntactic information (table layouts, units, encoding schemes, etc.) and semantic information (what an element ”means”, where it comes from, how it’s produced, how it’s used, etc.)[3]. For example, migrating payroll functions would require descriptions not only of the formats of employee records in the old HR system, but also descriptions of what constitutes a ”full-time” employee for benefits calculations, (e.g., appearance on the employee roster? having a particular form of employee number? reporting forty hours of time worked? Are different definitions of ”full-time” employee used for different purposes?)

Similarly, as parts of the functionality are migrated, new and old code modules must interoperate. For example, old modules for computing employee paychecks may need to interoperate with new check-cutting modules. Later in the migration, the old paycheck computational modules may be replaced and all will have to continue to interoperate with the old time reporting system. As another example, suppose the migration of the payroll system proceeds by geographic region. As each region is converted, payroll data from the migrated regions must be combined with similar data from any remaining regions for computations done in the (new) tax withholding modules. Support for interoperability requires metadata describing the components of the legacy and the migrated systems.

[1] Assuming, of course, that the system information was not loaded into the repository during earlier development or maintenance.

The repository can store descriptions of interfaces and relationships among the components of the old and new systems (e.g., between data elements and programs, or among software components that invoke one another) as well as descriptions of the semantics of the old and new data (and code) in the various migration increments. The repository also supports management of metadata describing software configurations, data flows, error conditions, and job structures and schedules. This could be used, for example, to determine when needed data has been updated or to obtain the transaction management requirements of the interoperating components.

Process management. Management of the migration process can be supported by metadata repository tools that track and control the configurations that form each of the increments and the changes to components from one increment to the next. Each release in a migration must be configured in such a way that necessary relationships between migrated and legacy system functionality and databases are preserved through the various versions of the migrated system, and the retirement of legacy components. For example, suppose that part of the migration process replaces project codes with a new coding scheme and that, in at least one system release, the project accounting modules use the old code and the time reporting modules use the new codes. This release cannot use a common project validation scheme, and historical records will include different code types. Managing the migration will require recording when the code changes occur, and mappings of modules to validation schemes in each version of a release. Testing procedures and error-correction mechanisms will also need to be aware of the various configurations. In addition, reporting systems will need to know the coding scheme changes to properly annotate reports.

Resolving process management issues depends on having information that describes the data and software components, and the business processes they implement. The information will change as the migration process unfolds, so resolving the issues requires not only up-to-date descriptions for each release version, but information about the process of change-when changes occur, the nature of the changes and the testing those changes trigger. In effect, what are needed are descriptions of the elements and the relationships among the elements, as well as descriptions of the changes in the elements and the relationships among the changes. Repositories generally provide versioning and configuration management tools that can document the contents of releases at each stage, thus maintaining an historical record of the migration. Moreover, they provide impact analysis tools that can then determine what elements were affected by each change and predict what will be affected by proposed ! ! changes.

4 Conclusions

Metadata repositories have long been used by CASE tools to store and manage descriptions of system components, and by data administrators to document information stores. More recently, they are being used to support integration of various tools, databases and applications, and their use is expanding to managing metadata for many more applications. We have shown how repositories can also be used to facilitate legacy migration by managing information about the existing and target systems and by providing tools to obtain, store, and analyze this information. As in new development efforts, design information in the repository can be used to generate some code or for decision-making in later stages of a migration. In addition, this information may be used to generate test cases and the repository can be used to track the testing process and associated scripts.

Metadata repository technology also helps to address specific issues of data transformation in the context of legacy migration:

1. Identifying appropriate sources of needed data and what transformations will be required to use that data in the migrated system is supported by data extraction tools and tools to assist with analyzing the metadata. The repository also provides a location for recording the relationships and transformations determined by a designer.

2. Interoperability among remaining legacy data (and systems) and migrated data is supported by a shared

model of data represented in the repository. In addition, descriptions of interfaces, component relationships, invocations, etc. can be stored in the repository, analyzed and used in designing and executing the integration.

3. Management of the migration process itself can be supported by metadata repository tools that track and control both the configurations that form each of the increments and the changes to components from one increment to the next.

A major advantage of using repository technology for storing metadata from a migration is the availability and usability of that information subsequent to the migration. For example, the information is useful in determining when elements of the legacy can be retired. Repository impact analysis tools can trace from new modules and data back to the legacy systems to see whether particular functions have been completely replaced (or replaced sufficiently) by the new systems. In addition, there is a continuing need for data from the migrated legacy to be combined with new data. The information gathered in the repository about the various systems and the migration itself can be used to compile accurate historical reports. Also, information gathered during the migration can be used in further maintenance of the system (correcting errors, new releases, adding enhancements, etc.) and may be useful in planning other migration projects.

Current applications of repository technology in software engineering typically address a single lifecycle stage or sequence of stages. Applications of repository technology to legacy migration tend to focus more broadly because they must include both the legacy and new systems and the components of each change as the migration process proceeds. However, they too tend to address only lifecycle stages directly related to the migration. Both applications would benefit by using the repository to obtain information about the existing environment and to store information during the project. We believe that significantly more benefits will accrue to the enterprise when metadata is shared across all stages of the lifecycle and across all projects. Our current work aims to engineer such complete information asset management using a repository.

References

[1] ASAP World Consultancy, Using SAP R/3. Que Corporation, 1996.

[2] S. Heiler, "Semantic Interoperability," ACM Computing Surveys, 27(2), 1995.

[3] V. Ventrone and S. Heiler, "Semantic Heterogeneity as a Result of Domain Evolution," SIGMOD Record, December, 1991; reprinted in Multidatabase Systems: An Advanced Solution for Global Information Sharing, A. Hurson, M. Bright, and S. Pakzad (eds), IEEE Computer Society Press, Los Alamitos, CA, 1993.

[4] J. Q. Ning, A. Engberts and W. Kozaczynski, "Automated Support for Legacy Code Understanding," in Communications of the ACM, 37(5), May 1994.

Independent, Open Enterprise Data Integration

Joseph M. Hellerstein, Michael Stonebraker, and Rick Caccia
{jmh,mike,rick}@cohera.com

Abstract

Database researchers and practitioners have long espoused the virtues of data independence. When logical and physical representations of data are separated, and when multiple users can see different views of data, then the flexibility of usage, evolution, and performance of a database system is maximized. This tenet has been lost in the marketing crush of Data Warehousing, which prescribes a tight coupling of physical representation and high-level usability.

In this paper we describe the Cohera Federated DBMS, which reintroduces data independence to the heterogeneous databases present in today's enterprises. Cohera's physical independence features provide a scalable spectrum of solutions for physical design of enterprise-wide data. Its logical independence features remove the distinction between data transformation and querying, by using industry-standard SQL99 as a unified open conversion framework.

1 Introduction

One of the major challenges and opportunities in corporate computing today is to facilitate the integration of data from multiple sources. As is well known, large enterprises have valuable information stored in a number of systems and formats: multiple relational databases, files, web pages, packaged applications, and so on. The reasons for the mix of technologies are also well known, including mergers and acquisitions, disjoint organizational structures, and the difficulty and expense of migrating from legacy systems.

The integration challenge has not escaped the attention of database vendors and researchers, and over the last decade there has been a flurry of effort to develop and market techniques for *data warehousing*. The idea of data warehousing is simple: in order to integrate data from multiple sources, the data is extracted from these sources, transformed into a common schema, and loaded into a single, unified database for the enterprise.

1.1 Warehousing: Dousing the Flame of Data Independence

From the dawn of relational databases in 1970, database wisdom has argued and repeatedly demonstrated that *data independence* is key to the success of large, long-lived information stores. Codd originally justified the relational model by arguing that it

> provides a means of describing data with its natural structure only - that is, without superimposing any additional structure for machine representation purposes. [This can] yield maximal indepen-

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

dence between programs on the one hand and machine representations and organization of data on the other. [Codd70]

Today, data independence is typically decomposed into two aspects. Physical data independence divorces the storage of data from its logical representation, maximizing the ability to tune and evolve the physical storage of data without affecting existing applications. Logical data independence divorces the underlying logical representation from the many different views presented to applications or users, maximizing the ability to customize the presentation of information for purposes of convenience or security.

The lessons of data independence apply directly to an enterprise with its many data sources. The task of integrating multiple data sources should be divorced from the specifics of the physical integration - data should be able to move and replicate in the enterprise without affecting applications. Similarly, the logical representation of the data should be malleable, to allow different users to see different views of all the data in the enterprise. This is particularly important in global enterprises, where different users will speak different languages, use different currencies, support different business models, and care about different aspects of the enterprise.

Somehow, the message of data independence was lost in the crush of data warehouse marketing. Advocates of warehousing heard customers' desire for a unified *access* to enterprise-wide data, and addressed it with uniform *storage*: an inflexible centralized layout of all enterprise-wide data in a single system. The result is a chain of expensive, brittle software that is hard to configure, manage, scale and evolve. Warehouses and their associated tools are unable to provide the physical data independence that allows flexible storage. Moreover, their, proprietary data transformation frameworks are not reusable in query processing, which minimizes the opportunity for logical data independence to provide different views to different users.

1.2 Federated Database Systems: Open Systems for Integration

By contrast, a good *Federated Database System* (FDBS) provides full data independence. It ensures flexibility in how the data is stored, replicated, and represented across the enterprise, allowing graceful and cost-effective evolution of enterprise-wide databases. It also allows for multiple logical views of an enterprise's data, including user-customizable conversions of types, policies, and schemas. An FDBS extends Codd's vision of data independence to the federation. Note that a warehouse is a point in the FDBS design space, where all data is physically unified, and converted into a single logical representation of types and tables. (Figure 1).

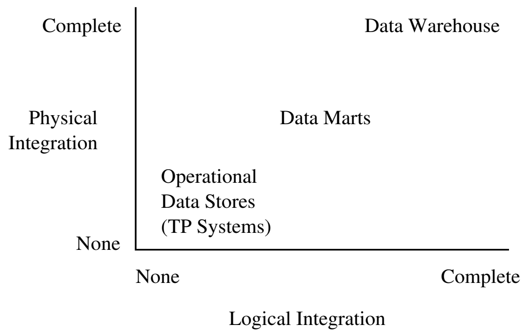

Figure1: The logical and physical integration spectra. Without data independence, current solutions only offer points in the space. Cohera provides full logical and physical independence, offering solutions spanning this space.

In this paper we contrast the integration technology of the Cohera FDBS with the available solutions in the warehouse space. We highlight the flexibility of the FDBS approach in enterprise-wide physical database design, and show how the high-performance techniques of data marts and warehouses can be leveraged as component technologies in the federated space. We also consider the effects of proprietary warehouse transformation packages on logical data independence, and contrast them with the open standard of SQL99.

2 Background: Buzzword Bingo

The research community has assimilated much of the warehousing lingo crafted by the business world. Before discussing the relevant issues, it is worthwhile to step back and have a fresh look at the terminology.

Data Warehouse: Consultant and industry pundit Bill Inmon is widely credited with coining this term. He meant by it a single, large system in which all an enterprise's data over time would be stored. Most database researchers, looking at a data warehouse engine, would instantly identify the software as nothing more or less than a relational database management system, albeit one tuned for a particular workload. *It is the workload that distinguishes warehouses from standard DBMSs*: warehouses are used in an append-only, query-mostly manner. As a result, they include support for high-volume, large-scale decision-support queries, with a focus on techniques like bitmap indices [OQ97], pre-computed results (e.g. materialized views [GM95]), aggressive query rewriting [LPS+98], and parallel query processing [DG92]. Warehouse databases are often laid out in denormalized ("star" or "snowflake") schemas, which favor high performance on queries, and low performance on updates.

Data Mart: Many organizations have found it too expensive and difficult to construct and maintain a complete data warehouse. In the absence of a single enterprise-wide data warehouse, the industrial consultants and vendors recommend constructing smaller *data marts*. These are relational database systems that consolidate the information in a particular area of an enterprise. Like warehouses, marts are targeted to append-only, read-mostly workloads. Data marts are essentially small data warehouses, and use the same technologies for enhancing query processing. Since marts are smaller than warehouses, they can also exploit some less-scalable but useful solutions, like the pre-computed multidimensional disk layouts used by some OLAP tools [DNR+97].

Who cares? Data warehouses and marts are nothing more or less than SQL database systems. So why the hype? Why do the database vendors discuss their "warehouse solutions" separately from their relational engines, when the software being used is identical? And what do warehouses and marts have to do with integrating enterprise-wide data? These questions are best answered by the marketing departments and consulting companies, of course. But the technical issues are really twofold.

First, the sheer scale of data warehousing has pushed the envelope in the design of relational database engines. In order to process decision-support queries over terabytes, vendors and researchers have been forced to deploy a whole host of the query processing technologies developed in research and industry over the years. The resulting systems represent impressive technical achievements, and have also produced well-known payoffs for some customers, particular large retailers.

The second technical issue that arises in data warehousing is the challenge of data integration. In the next section we say more about this issue, and comment on technology options for addressing it.

3 Approaches to Data Integration

Any enterprise of even moderate size has multiple sources of data: accounting systems, personnel systems, customer databases, and so on. Typically these data sources are managed by separate pieces of software. In large enterprises, there are often multiple data sources for each task, distributed geographically across the globe.

There are a number of products available for integrating the data in an enterprise. In this section we provide an overview of the extant approaches to data integration, and argue the merits of replication for physical independence, and SQL99 for logical independence.

3.1 ETL: Extract-Transform-Load

A number of vendors sell so-called Extract-Transform-Load (ETL) tools, including Informatica, Sagent, Ardent and Platinum. ETL tools extract data from underlying data sources via both native DBMS gateways (e.g. from relational vendors, ISAM, etc.) and via standard interfaces like ODBC; they then load the data into a warehouse Typically, an ETL tool also provides a facility to specify data transformations, which can be applied as the data is being extracted from the data sources and loaded into the warehouse. Most tools come with a suite of standard transformations (e.g., substring search/replace, units conversion, zip-code +4 expansion), and some extensibility interface for users to write their own transformations in a procedural language like C, C++ or Basic. Transformations can typically be composed into a pipeline via a scripting tool or graphical interface.

A main problem with ETL tools is that they were designed to solve a niche problem - warehouse loading - and are not useful for general data transformation. In particular, they have proprietary transformation extensibility interfaces: customers must program in to a specialized API or in a specialized language. Beyond imposing a learning curve on IT departments, the tight-focus philosophy prevents reuse. Transformations developed for the ETL tool cannot be reused in other data-conversion settings, particularly in ad hoc queries.

In essence, *ETL tools display a lack of logical data independence*: warehouse data can be transformed only during a physical load, not during querying. Among other problems, this prevents users from having different logical views of warehouse data. For example, once worldwide sales data is converted into dollars in the warehouse, French users cannot retrieve their local sales in Euros.

3.2 Replication Servers

Data replication servers evolved before the advent of warehousing, and typically have different features than ETL tools. In essence, replication servers perform only the extract (E) and load (L) facilities of ETL tools, but in a transactionally secure and typically more efficient manner.

The goal of a replication server is to ensure that transactions committed in one DBMS are reflected in a second DBMS. This can be used in a warehouse environment, to ensure that updates to the operational stores are reflected in the warehouse. It can also be used to support more flexible replication than the "single warehouse" model. For example, replication can be used in a master-slave mode to provide warm standby copies of important sites or tables to ensure high availability. Master-slave mode can also be used to improve performance, by allowing queries in a distributed enterprise to run on "nearby" copies of data. Replication can also be used in peer-to-peer mode, allowing updates to happen in multiple sites, with consistency ensured by conflict resolution schemes. Unlike ETL tools, replication servers do not work under an assumption that all data is loaded into a single warehouse. As a result, replication servers can be used to implement a flexible spectrum of physical replication schemes.

Replication servers are useful tools for flexible data layout, and hence can coexist gracefully with federated systems that support physical data independence. Since replication servers do no data conversion, they provide no solutions for logical data indepence.

3.3 FDBMS and SQL99: Scalar and Aggregate Functions

A federated DBMS extracts data from underlying stores on demand, in response to a query request. Data transformation is also done on demand by the FDBS. Transformations can be specified in the standard relational language, SQL99, extended with user-defined scalar and aggregate functions. Scalar functions operate a record at a time, taking a single record as input, and executing code to produce a modified record as output. In essence, scalar functions provide the utility to define simple conversions and combinations of columns. As an example, consider the following simple view that maps a table of customers from one data source, doing simple conversions: changing names into a single field of the form 'Lastname, Firstname', names of states into a standard postal code, and zip codes into the zip+4 format (user-defined functions in **boldface**):

SELECT **concat**(**capitalize**(lastname), ", ", **capitalize**(firstname)) AS name,
address, city, **twoletter**(state) AS state, **zipfour**(zip) AS zip
FROM customer;

This simple example demonstrates data conversion using user-defined scalar functions, producing one record of output for each record of input.

By contrast, aggregate functions are used to "roll up" a number of rows into a single result row. With user-defined aggregation and grouping, SQL99 allows for queries that collect records into affinity groups, and produce an arbitrary summary per group. For example, consider the following view, which uses a heuristic to remove duplicate entries from a mailing list (user-defined scalar functions in **boldface**, user-defined aggregates in *italics*):

SELECT	*std_last*(lastname) AS last, *std_first*(firstname) AS first, *typical_order*(order) AS std_order
FROM	customer
GROUP BY	address, **Name_LCD**(firstname, lastname);

In this example, the Name_LCD scalar function generates a "least common denominator" for a name (e.g. "Brown, Mr. Clifford and "Brown, Cliff are both reduced to "Brown, C"). All people at the same address with the same "LCD" name are grouped together. Then for each apparently distinct person in the cleansed list, a canonical output name is generated and a "typical" representative order is output based on all the person's orders (e.g., based on a classification algorithm, or a domain-specific rollup hierarchy).

Used aggressively, user-defined grouping and aggregation provide powerful features for data transformation: they can be used not only to convert units or replace substrings, but to *accumulate evidence* based on a number of observations, and generate a *conclusion* based on the evidence. In essence, SQL's grouping facility provides a means to gather evidence together, and SQL's aggregation facility provides a means to specify conclusions that can be drawn.

The extensibility features of SQL99 present a natural, open interface for data transformation. Unlike the proprietary APIs and scripting languages of ETL tools, SQL is the standard interface for combining and transforming sets of data. With the extensibility of SQL99, these features can be used aggressively to do custom domain-specific transformation. More importantly, user-defined scalar and aggregation functions can be reused in other SQL-based applications, which lets users leverage their investment in transformation logic for subsequent ad hoc querying. *Logical data independence requires that the transformation language and the query language be unified*; hence SQL99 is the only natural solution.

4 Cohera: Flexible, Scalable Enterprise Data Federation

Cohera is a next-generation FDBS, based on the Mariposa research done at UC Berkeley [SAL+96]. Cohera presents the best features of an FDBS: it provides the expressive transformation power of SQL99 along with a full spectrum of data independence achieved by coordinating with multiple existing replication tools. Cohera goes well beyond the state of the art in federated databases, however; the economic model pioneered in Mariposa provides the only available solution for serious *performance scalability* and *administrative scalability*.

We proceed to discuss the advantages of FDBSs for enterprise integration, and of Cohera in particular.

4.1 FDBS + Replication $>>$ ETL + Warehousing

Cohera recognizes that the key to physical data independence in a federation is to allow replication to be carried out aggressively and incrementally. This means that the FDBS should operate on any replica of a table, or (de)normalized replica of a schema. In order to facilitate this, Cohera interoperates seamlessly with a variety of replication tools, including market leaders from Sybase and Oracle. Cohera also includes a built-in replicator that can provide extra efficiency for certain replication tasks Regardless of the replication tool used, Cohera can take advantage of whatever replicas are available in the federation, and choose the most efficient replica dynamically for each query.

By contrast, ETL and Warehousing provide a single point on the physical and logical spectra, breaking the notion of data independence. As a result, warehouse shops need to invest in a centralized computing environment large enough to hold all of an enterprise's data. This requires an enormous investment in equipment and software, and in the skilled administration required to keep such installations running. As one example from the real world, a large telecommunications firm plans employs warehouses and finds good return on investment from implementing them. However, before taking on the time, expense, and headcount to implement and support another warehouse, the firm wants to understand whether the data integration will be useful. A federated database will be used to deliver a logical view of multiple sources without requiring physical co-location. In effect, the FDBS delivers prototyping and migration facilities even for warehousing advocates.

An additional benefit of an FDBS over warehousing is the timeliness of data: an FDBS can provide access to live data from operational stores, rather than outdated data fed into a warehouse once a day, week or month. The tradeoff of timeliness and the impact on operational stores is one that can be made flexibly, depending on query vs. transaction load; this is another benefit of the data independence afforded by FDBSs.

Finally, note that since FDBSs and replication use standard SQL99 as their transformation scheme, existing tools and techniques for materialized views can be integrated with the replication process. This allows FDBSs to span the the logical/physical spectra of Figure 1, by allowing not only tables but also views (with SQL99 transformations!) to be either logical or physical.

4.2 Cohera $>>$ 1st-Generation FDBS

Like Mariposa, Cohera integrates underlying data sources into a *computational economy*, where data sources cooperate in distributed query processing through a metaphor of "buying" and "selling" their services to each other. Unlike Mariposa, Cohera explicitly focuses on federating heterogeneous systems.

When an SQL query is submitted to a Cohera site, a module at that site called the *contractor* translates the query into a single-site execution plan. The contractor must consider each operation in the execution plan - e.g. scanning a table, or performing a join - and choose the most efficient data source to execute that operation. To do this, it solicits *bids* for each constituent part of the plan: for example it might solicit bids for the operation of scanning a table called "sales". This bid is sent to other Cohera sites (*bidders*) that manage their own underlying stores. If a given bidder site manages a database with a copy of the "sales" table, it might choose to bid on the operation. In this case it uses its own local rules to place a cost on the operation, and it ships its bid to the contractor. The contractor collects bids for all the individual operations, and constructs a distributed query plan that minimizes the cost of the whole query.

This economic model provides enormous benefits over traditional FDBS designs. First, the economic model ensures *local autonomy* for managers of databases in the federation, who can control bidding policy based on local constraints (e.g., "during 9-5 only bid on federation queries from the office of the CEO".) Second, the economic model gives local managers the facility and incentive to participate in enterprise-wide *load balancing*: a standard bidding policy is to have the bid price be proportional to the product of work and local load average, which naturally spreads work to under-utilized servers. Third, the economic model provides *scalable performance*, by distributed the query optimization and scheduling process. First-generation FDBSs had centralized optimizers that needed to know the state of all machines in the federation in order to operate; this prevented scaling beyond a small size. Finally, federating the query optimization process provides *administrative scalability*, by allowing local administrators to set costs and policies based on dynamic properties (load balance, time of day, installation of new hardware, etc.); the centralized optimizers of first-generation FDBSs limited the autonomy of local administrators, since the centralized optimizer had to have control to work correctly. A lack of autonomy requires consensus among database administrators, which is politically (and often geographically) infeasible in an organization of any size.

As a final contrast between Cohera and earlier systems, note that the "open marketplace" in Cohera allows sites to be added and deleted over time. Coupled with replication, this facility gives an enterprise an incremental path to migrate along the spectrum of Figure 1. Note that a warehouse is not ready for production until *all the data is loaded*. By contrast, a federation can start out as a loose affiliation of a few operational stores, expand to include a denormalized query site (e.g. a data mart) to enhance some queries, and eventually grow to span the enterprise. This minimizes the management and cash flow needed at any point in time. It also allows for incremental upgrade of components, rather than a complete swap of old iron for new iron. Note that incremental upgrade means that you are always buying a little of the latest hardware at the lowest price, rather than buying a lot of today's hardware at today's prices to plan for tomorrow's capacity.

5 Conclusion

Enterprise data integration is one of the most pressing problems in IT today, and a subject of much research interest. Cohera's solution to this problem is to ensure full data independence: Cohera provides a flexible, scalable infrastructure for enterprise-wide data storage, and a unified transform/query interface based on the open standard of SQL99. Cohera's unique scalability is the result of its economic computing model, which scales both in performance and administrative complexity.

Bibliography

[Codd70] Codd, E.F. "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM 13(6):377-387, 1970.

[DG92] DeWitt, D. J. and J. Gray. "Parallel Database Systems: The Future of High Performance Database Systems." Communications of the ACM 35(6): 85-98, 1992.

[DNR+97] Deshpande, P., J. F. Naughton, K. Ramasamy, A. Shukla, K. Tufte, amd Y. Zhao. "Cubing Algorithms, Storage Estimation, and Storage and Processing Alternatives for OLAP". Data Engineering Bulletin 20(1):3-11, 1997.

[LPS+98] Leung, T.Y.C., H. Pirahesh, P. Seshadri and J. M. Hellerstein. "Query Rewrite Optimization Rules in IBM DB/2 Universal Database", in Stonebraker and Hellerstein (eds.), Readings in Database Systems, Third Edition, Morgan-Kauffman, San Francisco, 1998.

[GM95] Gupta, A. and I. .S. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications."' Data Engineering Bulletin, 18(2), June 1995.

[OQ97] O'Neil, P. E. and D. Quass. "Improved Query Performance with Variant Indexes". Proc. ACM SIGMOD Conference, Tucson, June, 1997, pp. 38-49

[SAL+96] Stonebraker, M., P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu. "Mariposa: A Wide-Area Distributed Database System." VLDB Journal 5(1): 48-63 (1996)

Supporting Retrievals and Updates in an Object/Relational Mapping System

Jack Orenstein
Novera Software, Inc.
jack@novera.com

1 Overview

Novera Software, Inc. produces jBusiness, a pure-Java application server [NOVE]. Enterprise Business Objects (EBO) is an object/relational mapping system, which is part of jBusiness. This paper describes some of the transformations that EBO uses to provide transparent, high-performance database access to middle-tier Java applications.

2 Business Objects

An EBO application manipulates business objects. A *business object* is an object representing some concept relevant to the application, whose state is obtained from an underlying relational database. Much of the code of an application can be expressed in terms of business objects. Other code relies on the EBO application programming interface (API), and a collection library.

This section describes business objects, how they are mapped to Java objects, and how these objects are mapped to tables in a database. For more information, see [NOVE98].

2.1 Object Model

The EBO object model is extremely simple. Objects have identity. Objects have scalar properties, and may participate in one-to-one and one-to-many relationships.

A *simple scalar property* is mapped to a single column. A *computed* property is mapped to a SQL expression, e.g. QUANTITY * UNIT_PRICE. An *aggregate* property is mapped to a SQL expression containing an aggregation operator, e.g. SUM(QUANTITY * UNIT_PRICE). Each property gives rise to methods for getting and setting the property's value. "Set" methods are useful even for computed and aggregate properties as they allow the application to reflect changes due to actions by the application itself.

Foreign keys are mapped to one-to-one or one-to-many relationships. When a relationship is realized in Java, there are methods in the two participating classes. The methods generated depend on the cardinality of the relationship. For example, consider the one-to-many relationship between Employees and Departments. EBO will generate the methods Department.getEmployees, which returns a collection, and Employee.getDepartment,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

which returns a Department, for navigating the relationship in either direction. The methods Employee.insertEmployee, Employee.removeEmployee and Department.setDepartment modify the relationship.

Collections play two roles in EBO. First, collections are used to represent the "many" side of a relationship, as described above. Collections called *extents* are also used to represent the set of all instances of a business object class (in a database). The extent of a class can be obtained by calling the business objects extent method. For example, Employee.extent() returns the set of all Employee objects. Invoking extent() does *not* cause the retrieval of all the data in the database required to materialize all the Employee objects; it is merely a surrogate. Operations on extents result in the execution of SQL commands as described below. Because an extent is a collection, collection operations can be applied to it.

2.2 Defining Business Objects

The *schema mapper* is a GUI which assists the user in defining business objects. The schema mapper starts by connecting to a database and letting the user select tables of interest. Definitions of these tables are read in, and from these definitions, business object object definitions are created. The schema mapper generates default definitions, which can then be extended and customized. The schema mapper also lets the user customize the translation of business object definitions into Java object definitions.

In general, a business object combines one or more tables through joins, and projects selected columns. When defining a business object, one of the participating tables must be specified as the *root table*. Intuitively, the business object represents the same real-world entity that the root table does. Internally, the root tables primary key is used to determine the business objects identity.

For example, suppose the database has tables PRODUCT and CATEGORY, in which each PRODUCT has a CATEGORY. We might want to define a Product business object which is based on PRODUCT but includes some information from CATEGORY, e.g. CATEGORY_NAME. PRODUCT is the root table of Product. Product can have a CategoryName property based on CATEGORY.CATEGORY_NAME.

The schema mapper offers a simple way to turn a set of tables into a set of business objects. Each table is turned into a root table for some business object. Columns are turned into properties, and foreign keys are turned into relationships. These default definitions can be modified in a number of ways.

Once the business object and Java object definitions have been defined, the schema mapper can be directed to generate its output. The output comprises a set of Java sources, "class files" (resulting from compilation of the sources), and a *schema map*. The schema map represents all the information read and created by the schema mapper. The transformations carried out by the EBO runtime are guided by the schema map, as described in the following sections.

3 Simple Retrievals

For each business object, a method named *find* is generated to locate the object given its primary key, e.g. Customer.find(419). The implementation of find first checks the EBO cache to see if it contains the object, and if it is non-stale. A database query is executed if no such object exists. The SQL is generated by simply adding primary key selection clauses to the business objects view definition. *Navigation* describes the invocation of a business objects method which returns either another business object, or a set of business objects. These are the *get* methods associated with relationships. As with *find*, a database query is issued only if the required object(s) are not present in the cache.

4 View Expansion

4.1 View Table Queries and Base Table Queries

Consider the tables PRODUCT and CATEGORY, and the Product business object, (described in section 2.2). In the terminology of the EBO implementation, PRODUCT and CATEGORY are *base tables* - they are defined in the database. EBO also has the concept of a *view table* which corresponds to the view definition mapping Product to PRODUCT and CATEGORY. (When we use the term view we mean a definition stored in the EBO schema map; we never refer to a view definition in the database. A view in the database is just another (base) table.)

There is one subsystem, the *sql* subsystem, that has an API for creating SQL queries, adding restrictions, join terms, group by clauses, order by clauses, and so on. Those parts of EBO that translate retrieval operations into SQL use this API to build up SQL queries *in terms of view tables*. The sql subsystem is responsible for expanding the view definitions to come up with SQL queries involving base tables.

4.2 Outerjoin Translation

Now assume PRODUCT and CATEGORY tables are connected by a foreign key from PRODUCT.CATEGORY_ID to CATEGORY.CATEGORY_ID. Because the Product business object includes CATEGORY.CATEGORY_NAME, Product's view definition includes a join between the PRODUCT and CATEGORY, across the foreign key. To provide reasonable semantics to Java applications, the join must be an outer join. For example, iteration over the extent should locate all Products, including uncategorized ones, (i.e. PRODUCT rows with CATEGORY_ID = NULL). Such products are dropped by an inner join between PRODUCT and CATEGORY.

Outerjoin support by relational database systems is quite uneven. For this reason, we do not rely on native outerjoin capabilities. Instead, we simulate outerjoins using other techniques, e.g. a UNION of subqueries. This is quite expensive compared to an inner join, and in many situations the extra work is unnecessary maybe there really arent any uncategorized products, and the user knows this. To avoid (simulated) outerjoins when possible, the schema mapper allows the user to declare, for each foreign key, that certain joins can be inner joins for the purpose of materializing a business object.

4.3 Update Translation

Updates are handled differently due to the usual difficulties in updating through views. Whenever SQL update code is generated, (using the INSERT, DELETE or UPDATE commands), the updates are expressed directly in terms of base tables; i.e., the sql subsystem is not used for view expansion.

4.4 Query Language

EBO provides a Java-like query language resembling Orion [KIM89] and ObjectStore query languages [OREN92]. It was designed to be familiar to Java programmers rather than SQL programmers, and this is reflected in the syntax. It does, however, have capabilities similar to several SQL extensions (e.g. OQL [ODMG97]).

Example 1: Find Customers whose company name matches a given pattern.

```
Query company_name_query = DbBlend.createQuery
(``Customer customer; String pattern:  ''+
``customer.getCompanyName.like(pattern)'');
```

This query declares a Customer range variable, customer, and a free variable containing the pattern. The predicate locates those companies whose name matches the pattern. getCompanyName is a method for accessing the CompanyName property. (DbBlend is an earlier name for EBO. EBOs API still refers to this older name.)

Example 2: Find Customers still waiting for orders placed more than one year ago.

```
Query major_customers = DbBlend.createQuery
(``Customer c; Date order_date:  c.getOrders[''+
`` Order o:  o.getOrderDate ≥ order_date && ''+
`` o.getShipDate == null]'');
```

This query contains a subquery which looks for qualifying Orders. A Customer appears in the result if it has at least one qualifying Order.

5 Basic Query Translation

An EBO query is parsed, creating a parse tree in which leaves correspond to range variables, free variables, and literals; and internal nodes represent operators such as navigation (the "dot" operator), comparisons, boolean operators, and nested queries. Then a bottom-up scan of the parse tree is performed. At each node, some portion of a SQL query is created, using the API of the sql subsystem. The results of this generation are associated with the node. When this traversal is complete, the root of the tree contains a complete SQL query.

6 Prefetch

Consider an order entry GUI that deals with Customers, Orders, OrderDetails, and Products. Given a Customer's primary key, the GUI might need the Customer's Orders, each OrderDetail of each Order, and all Products referenced in any Order. All this information can be retrieved by navigation, using the methods Customer.getOrders, and Order.getOrderDetails, OrderDetail.getProduct. EBO provides an alternative. A description of a path can be supplied when the query is created. If this is done, then the SQL generated is modified to *prefetch* the related objects along the path specified. Without prefetch hints, the number of SQL queries generated by the navigation is proportional to the number of objects retrieved. With prefetch, different queries are issued, and the number of queries is proportional to the length of the prefetch hints path description. (In the example above, there will be exactly four queries, one for each class: Customer, Order, OrderDetail and Product.) For example, here is code that locates Customers whose company name contains the string "IBM", and prints descriptions of their Orders:

```
Query ibm_customer_query = dbblend.createQuery
(``Customer c:  c.getCompanyName.like(\''%IBM%\``)'');

for (ibm_scan = Customer.extent().elements(ibm_customer_query);
ibm_scan.hasMoreElements();) {
Customer customer = (Customer) ibm_scan.nextElement();
for (order_scan = customer.getOrders().elements();
order_scan.hasMoreElements();) {
Order order = (Order) order_scan.nextElement();
print_order(order);}
}
```

Assuming execution begins with an empty cache, there is one query to the database to fetch the qualifying Customers, and then another query for each Customer to get that Customer's Orders. The Order queries would look something like this:

```
SELECT o.order_id, o.order_date, o.ship_date
FROM ORDER o
WHERE o.customer_id = ?
```

The value of the free variable (denoted by ?) is the primary key of the current Customer.

Two modifications are required to use prefetch. First, the navigation path needs to be described:

1. Create a path description starting at Order
```
PathSet customer_to_order = dbblend.createPathSet(``Customer'');
```
2. Extend the path by navigation through getOrders to an Order.
```
customer_to_order.add(``getOrders(Order)'');
```

Then, add the path description as a second input to the createQuery method:
```
Query ibm_customer_query = dbblend.createQuery
(``Customer c:c.getCompanyName.like(\''%IBM%\``)'',
customer_to_order);
```

Now, when the query is executed, both Customers and Orders will be retrieved and placed in the cache. Instead of running multiple queries looking for the Orders rows with a given customer id, there will be one query to get the Orders for all qualifying Customers:
```
SELECT o.order_id, o.order_date, o.ship_date
FROM ORDER o, CUSTOMER c
WHERE c.COMPANY_NAME LIKE *%IBM%*
AND o.CUSTOMER_ID = c.CUSTOMER_ID
```
As a result, navigation from Customer to Order using customer.getOrders never needs to go to the database; the result is already in the cache.

PathSets provide a very easy way to tune performance. Application developers can develop their business logic using queries and navigation. When performance problems are detected, they can often be addressed by simply adding a PathSet to the queries, leaving the bulk of the business logic alone. If a query is created with a PathSet argument, then there is an additional phase in query translation. Following the creation of the initial SQL query a new query is generated for each step of each path represented by the PathSet.

7 Conclusions

EBO has proven successful in practice. Customers have quickly developed applications by mapping database schemas to business objects, writing applications in terms of business objects, and letting EBO take care of nearly all interaction with the database. Typical applications involve 100-150 tables, mapped to a smaller number of business objects.

8 References

[KIM89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk: Features of the ORION Object-Oriented Database System. OO Concepts, Databases, and Applications 1989: 251-282

[NOVE] Novera web site, http://www.novera.com.

[NOVE98] Developing Enterprise Business Objects, Novera Software, Inc., December 1998.

[ODMG97] The ODMG 2.0 Specification, Morgan Kaufmann, (1997).

[OREN92] Jack A. Orenstein, Sam Haradhvala, Benson Margulies, Don Sakahara: Query Processing in the ObjectStore Database System. SIGMOD Conference 1992: 403-412.

CALL FOR PAPERS

DATA Φ ENGINEERING

The 16th International Conference on

Data Engineering

Holiday Day Inn on the Bay, San Diego, CA, USA
February 28 - March 3, 2000
Sponsored by
IEEE Computer Society TC on Data Engineering

IEEE

SCOPE

Data Engineering deals with the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments. The 16th International Conference on Data Engineering will continue in its tradition of being a premier forum for presentation of research results and advanced data-intensive applications and discussion of issues on data and knowledge engineering. The mission of the conference is to share research solutions to problems of today's information society and to identify new issues and directions for future research and development work.

TOPICS OF INTEREST

These include (but are not restricted to):

- Advanced Query Processing
- Data Mining & Knowledge Discovery
- Engine Technology (Storage Management, Access Structures, Recovery, etc.)
- Multimedia & Digital Libraries
- New Departures (Mobile Agents, Embedded Services, Personalization, etc.)
- New Forms of Data (Spatial, Temporal, etc.)
- OLAP & Data Warehouses
- Replication, Availability, & Consistency
- System Administration, Ease of Use, and DB Design
- Web, Distribution, & Interoperation
- Workflow, Transactions, & E-Commerce
- XML & Metadata

IMPORTANT DATES

Abstract submissions (electronic in Ascii): June 9, 1999

Paper submissions (hardcopy or electronic): June 16, 1999

Web page has electronic submission details

Panel/tutorial/industrial submissions: June 16, 1999

Acceptance notification: October 15, 1999

Camera-ready copies: December 1, 1999

ORGANIZING COMMITTEE

General Chair:	P.-Å. (Paul) Larson, Microsoft, USA
Program Co-chairs:	David Lomet, Microsoft, USA Gerhard Weikum, Univ of Saarland, Germany
Panel Program Chair:	Mike Carey, IBM Almaden, USA
Tutorial Program Chair:	Praveen Seshadri, Cornell Univ, USA
Industrial Program Co-Chairs:	Anil Nori, Ventis, Inc, USA Pamela Drew, Boeing, USA

PROGRAM VICE-CHAIRS

Advanced Query Processing
Jeff Naughton, Univ of Wisconsin, USA

Data Mining & Knowledge Discovery
Sunita Sarawagi, IBM Almaden, USA

Engine Technology
Hank Korth, Lucent - Bell Labs, USA

System Admin, Ease of Use, & DB Design
Arnie Rosenthal, Mitre, USA

OLAP & Data Warehouses
Jeff Ullman, Stanford Univ, USA

Workflow, Transactions, & E-commerce
Hans Schek, ETH Zurich, Switzerland

XML and Metadata
Phil Bernstein, Microsoft, USA

Web, Distribution, & Interoperation
Donald Kossmann, Univ of Passau, Germany

Multimedia & Digital Libraries
Stavros Christodoulakis, Univ of Crete, Greece

Replication, Availability, & Consistency
Theo Haerder, Univ of Kaiserslautern, Germany

New Forms of Data
Beng Chin Ooi, National Univ of Singapore, Singapore

New Departures (Agents, Mobile, etc.)
H. V. Jagadish, U Illinois - Urbana-Campaign, USA

SUBMISSIONS

Paper, panel, tutorial, or industrial-program submissions must be received by June 16, 1999, 12:00 PM Pacific Time. Paper length must not exceed 20 pages in not smaller than 11 pt font.

For hardcopy submission, seven (7) copies should be sent to:

David Lomet
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399, USA
E-mail: lomet@microsoft.com

For electronic submission, please consult the conference web site at: **http://research.microsoft.com/icde2000**.

An abstract of no more than 300 words in ASCII must be submitted at the conference web site: **http://research.microsoft.com/icde2000** by June 9, 1999. The abstract submission must include the title of the paper, authors' names, an e-mail address of the contact author, a first and second preference among the twelve topical areas, and whether the submission is for the research paper, panel, tutorial, or industrial program.

PANELS AND TUTORIALS

The research and industrial track will be complemented by panels and tutorials. Proposals for each program should be sent to the same address for paper submissions (see there) by **June 16, 1999**.

INDUSTRIAL PROGRAM

The conference program will include a number of papers devoted to industrial developments, applications, and experience in using databases. Papers intended for this program should be clearly marked as industrial track papers at the time of submission.

Conference web site: http://research.microsoft.com/icde2000

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398