Model Management Through Graph Transformation

Guanglei SONG **Kang ZHANG** **Jun KONG**
Department of Computer Science, University of Texas at Dallas
Richardson, Texas 75083-0688 USA
{gxs017800, kzhang, jxk019200}@utdallas.edu

Abstract

Model management offers a higher level interface than current techniques for metadata management, and generic operators drastically reduce amount of programming for metadata applications. The interactive nature of generic model management operators inevitably demands an intuitive representation. This paper proposes a visual representation for model management operators based on graph transformation. Graph transformation formalisms, as the theoretic foundation of many visual programming languages, can formally represent model management operators by visual and intuitive expressions. By using graphical representations users can easily comprehend and manipulate the operators and desired outputs.

1. Introduction

Tremendous data is available in heterogeneous formats, such as relational database schemas and XML documents. Engineers who manage information systems usually need to design, integrate, transform, or evolve these application artifacts that are used to define data formats and are called *models* or *metadata*, such as ER models, relational, and XML schemas. Traditional approaches to implementing data applications need to program specifically for the corresponding metadata, i.e. using object-at-a-time primitives, which are hard to develop, adapt, and evolve in different contexts.

Model management is a new approach to metadata management that offers a higher level programming interface than current techniques [Ber00] and avoids object-at-a-time primitives. It aims at reducing the amount of programming needed for metadata intensive applications by treating models and mappings as abstractions that can be manipulated by generic operators. Generic model management treats these abstractions as bulk objects and offers high-level operators on various metadata applications [Ber03].

Automated model management operators require a considerable implementation effort or sometimes are simply not feasible. For example, schema matching is *ad hoc* in nature [Bun92], and depends on the real-world interpretation of the underlying data sources. Schema-matching tasks are typically performed manually, sometimes using a graphical tool [Rah01]. At best, some tools can detect matches semi-automatically – even minor name and structure variations lead them astray [Bun92]. Other operators that are based on schema mappings have similar problems. Model management inherently requires human decision making.

Currently model management operators are described and performed by underlying algorithms, which are mostly transparent to users, and there is no way for users to interact with the system to regulate the output. This implies that users have to comprehend and manually adjust the intermediate and final results. The complexity of data models and mappings often makes the task error-prone and time-consuming, and degrades the applicability of a model management system. The interactive nature of model management operations inevitably demands an intuitive representation of operators, and a graphical representation becomes a viable option. However there have been no formally defined graphical operators except informal descriptions and visualization of mappings.

Graph offers intuitive means for describing data models. Graph transformation, as the theoretical foundation of many visual programming languages, is capable of formally defining how graphs should be built and how they evolve. Furthermore, the operators on data models can be defined by graph transformation. The graphical representation of operators and their transformation are intuitive, and perfectly match the interactive nature of model management operators.

The recently developed Reserved Graph Grammar (RGG) formalism is powerful in expressing various types of diagrams, with a parsing complexity of polynomial time under a non-ambiguous condition [Zha97, Zha01b]. Based on the RGG, this paper presents a formal visual model management approach. The paper contributes to model management in the following aspects:

- A generic visual representation of data models and mappings. The representation provides a formal syntax definition and verification mechanism for data models and mappings.
- The first proposal that uses graph transformation to describe model management operators and provides intuitive interfaces for users to tune the result.

- An interactive model management environment for users to create and customize operators for metadata applications by editing the graphical description of operators.

This paper is organized as follows: Section 2 introduces model management operators. Sections 3 and Section 4 present a brief overview of the RGG and our model management framework. Section 5 proposes a graphical definition of data models and mappings. Section 6 applies graphical representations to model management operators. To demonstrate the parsing process of a graphically defined operator, Section 7 presents a parsing example. Related work is reviewed and compared in Section 8, followed by the conclusion in Section 9.

2. Model Management Operators

The main model management operators are briefly described as follows [Ber03]:

- Match – takes two models as input and returns a mapping between them.
- Compose – takes a mapping between models A and B and a mapping between models B and C, and returns a mapping between A and C.
- Diff – takes a model A and mapping between A and some model B, and returns the sub-model of A that does not participate in the mapping.
- ModelGen – takes a model A, and returns a new model B that expresses A in a different representation (i.e. data model).
- Merge – takes two models A and B and a mapping between them, and returns the union C of A and B along with mappings between C and A, and C and B.

These operators are applied to models and mappings as a whole, rather than to their individual elements. The operators are generic in the sense that they can be utilized for different kinds of models and scenarios.

Consider a typical example of building a data warehouse [Ber03]: Suppose we are given a mapping map_1 from a data source S_1 to a data warehouse S_W, and wish to map a second source S_2 to S_W, where S_2 is similar to S_1 (Figure 1). First we call Match(S_1, S_2) to obtain a mapping map_2 between S_1 and S_2, which shows where S_2 is the same as S_1. Second, we call Compose(map_1, map_2) to obtain a mapping map_3 between S_2 and S_W, which returns the mapping between S_W and the objects of S_2 corresponding to the objects of S_1. To map the remaining objects of S_2 to S_W, we call Diff(S_2, map_3) to find the sub-model S_3 of S_2 that is not mapped by map_3 to S_W, and map_4 to identify the corresponding objects between S_2 and S_3. We can then call other operators to generate a warehouse schema for S_3 and merge it into S_W. Comparing to programming the whole system for individual requirements, using model management reduces considerable programming effort by composing generic operators.

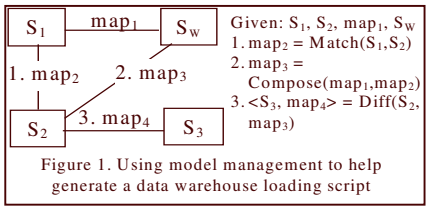

Figure 1. Using model management to help generate a data warehouse loading script

3. A Graph Grammar Formalism

The reserved graph grammar formalism (RGG) [Zha97] is expressed in terms of node-edge diagrams, a node is organized into a two-level hierarchy as illustrated in Figure 2. A large rectangle is the first level called a *super-vertex* with embedded small rectangles as the second level called *vertices*. In a node, each vertex is uniquely identified. The name of a super-vertex distinguishes the type of nodes, similar to the type of variables in conventional programming languages. Edges are used to denote relationships between nodes. Either a vertex or a super-vertex can be the connecting point of an edge. In addition to the structural information, the RGG provides a means of associating data to nodes in terms of attributes.

A RGG consists of a set of graph grammar rules, also called *productions*, each having two graphs that are called *left graph* and *right graph* as shown in Figure 2. A production can be applied to a given application (called *host graph*) in the form of an *L-application* or *R-application*. A sub-graph in the host graph is called a *redex* if it is isomorphic to the left graph in an L-application or to the right graph in an R-application. An L-application (R-application) to a host graph is to find in the host graph a redex of the left graph (right graph) of the production and replace the redex with the right graph (left graph) of the production. To identify graph elements to be reserved during the transformation process, we *mark* the vertex in a production graph corresponding to these elements by prefixing its label with a unique integer. If a vertex in a right graph is marked, it is allowed to be connected, in a host graph, to any node outside of the redex that matches the right graph. The marked vertex preserves its associated edges connected to the outside of the redex during parsing.

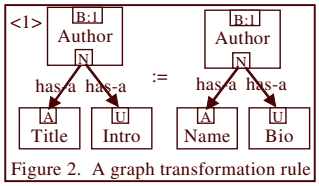

Figure 2. A graph transformation rule

Many data models are specified in diagrams with directed and attributed edges. To represent these data models, each RGG edge has two features: *direction* and *attribute*. An edge is denoted by a tuple *E(s, a, t)*, where s is the source node, t is the target node, and a is

the attribute of the edge. For the example in Figure 2, in the right graph of the production Author has two edges, i.e. edge (Author, has-a, Name) and edge (Author, has-a, Bio), where has-a describes the aggregation relationship between the source and target nodes. Attributes are hidden in most cases. A host graph is defined by a tuple *G (N, E, A)*, where N is a set of labeled nodes, A is a set of attributes, and E is a set of edges $E \subseteq N \times A \times N$. A production rule is defined by a tuple of left graph and right graph, i.e. *P (L, R)*, where $L, R \subseteq G$. A RGG consists of a set of production rules, i.e. GG = *{Pi}*.

The RGG offers a translation mechanism [Zha01a], i.e. graph *transformation rules* can specify the transformation from an input graph to a different graph as shown in Figure 2. On an input graph *S* and transformation rules *P*, one can apply *P* to *S*, i.e. *A (S, P)*, and expect an output *T*, where *T = A (S, P)*. Using the RGG transformation rules, one can visually program the transformation *R* of a graph *S* to another graph *T*.

A parser performs transformation by searching a redex in the host graph and replacing it with the left graph until no more redex can be found. To achieve high performance and avoid ambiguity we employ the selection-free parsing algorithm (SFPA) developed by Zhang *et al*. [Zha97].

4. Framework Overview

Various data models and mappings are specified by different syntaxes, which are mostly defined in natural languages in spite of some formal attempts [Rek97].

Our model management framework provides a formal visual representation of data models and mappings defined by the RGG as inputs of model management operators. It exploits graph grammars in defining the syntax of data models. The parser would detect any syntax violation of input data models and mappings. The formal definition also gives a foundation for defining various model management operators by graph transformation. Inputs to an operator are viewed as a set of host graphs compliant to the predefined abstract syntax.

In the framework, model management operators are specified at two levels, i.e. *specific operator* and *generalized operator*. A specific operator is a low level description of an operator on a specific input, and presents users a concrete image of the expected output and interface for tuning the result. A specific operator is automatically generated on specific inputs through a generalized operator that is at a high level abstraction, and can be applied to general inputs. The generalized operator graphically describes the algorithm used to transform the input to output of the operator, i.e. the algorithm is performed through a set of graph transformation rules. Since most model management operators require operations on mappings, i.e. results of the match operator, a generalized operator cannot produce perfect result without human intervention. But at a high level of abstraction, a generalized operator is hard to be adapted on specific inputs and is therefore necessary to cooperate with a customizable specific operator.

The two-level hierarchy of operators defines two levels of system-user interactions, i.e. *design level* and *operation level*. At the design level, experts of model management and graph transformation describe the algorithm of an operator by graph transformation rules, i.e. generalized operator. At the operation level, users, such as DBAs, perform metadata-intensive management tasks by adjusting and executing specific operators, which are generated automatically from generalized operators (the process will be described in detail in Section 6).

Figure 3 shows an overview of the framework, which embeds a set of predefined generalized operators. Users compose the operators by scripts or command line to construct metadata applications. According to the generalized operator the framework generates a set of specific rules as an interface to accept user's customization. During each step of the execution, users may adjust the customizable specific operators to obtain desired output rather than adjusting output directly which could be error-prone. After specific operators are parsed, a visual environment is generated, which produces final results of the operator.

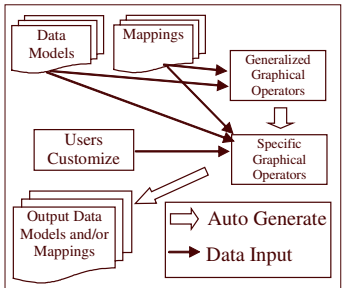

Figure 3. Framework overview

5. Graphical Representation of Models and Mappings

Graphs are used to represent a wide range of data models, such as XML schemas [Zha01a], DTDs [Ger99], and mappings [Ber03]. The RGG can formally define the syntax of graphs and thus the syntax of data models and mappings.

A data model contains a set of objects and various relations between the objects. An object could be an entity in ER models or an element in XML schemas, and a relation could be an "is-a" or "has-a" relation. Each object has an identity and type, and each relation has properties denoting its semantics, such as the min and max cardinality.

Our framework represents a data model, e.g. ER model, by a host graph in terms of a directed node-edge diagram. A node represents an object, and an edge denotes a relation. A node has a name and a type corresponding to the object it represents. The attribute of an edge defines the relation between the two connected objects.

Graphs of a kind of data model should be compliant to the syntax of that model. For example, two entities of an ER model cannot be connected directly. Such syntax is defined by graph grammar rules. The rules for XML Schemas can be found in our previous work [Son04] and similar rules can be constructed for other data models. With these rules, one can easily draw models under the syntax guidance of the RGG toolset [Zha01c].

A mapping, Map_{AB}, defines how models A and B are related [Pot03] as shown in Figure 4. Many proposals use graphical metaphors to represent schema mappings like in Rondo [Mel03], and Clio [Mil01]. These mappings are shown to the user as sets of lines connecting the elements of two schemas. Such kind of representation is simple but not as powerful as SQL view [Mad03] or that of Bernstein [Ber03]. SQL view is not a generic representation for mappings among heterogeneous data sources, such as XML schemas. On the other hand, mappings are structured instead of flat bi-directional, and hard to be described by simple two-way correspondences. The mapping structure described by Bernstein [Ber03] is an appropriate compromise, being generic and yet powerful for describing mappings.

Our framework represents mappings as special data models. A mapping has only one relationship type, i.e. has-a relationship, and three element types, i.e. *mapping element*, *reference element* and *helper element.* A *mapping element* specifies how the two referenced models' elements are related, such as equality, or similarity, such as node Equal in Figure 4. A *reference element* is a reference to the element of the two corresponding models, such as those nodes of Model A in Figure 4. The relationship between a mapping element and a reference element is denoted by a dashed line in this paper. A *helper element* is a make up element to represent extra semantics of a mapping. For example, Intros is a helper element indicating that Bio and Intro can be composed together to form a detailed and official description of Author as shown in Figure 4.

The syntax of mappings is defined by a graph grammar as in Figure 5, which includes five production rules. The first production shows that the initial state of the mapping is a mapping element. Production <2> shows that each mapping element can be connected to and has the has-a relationship with more than one

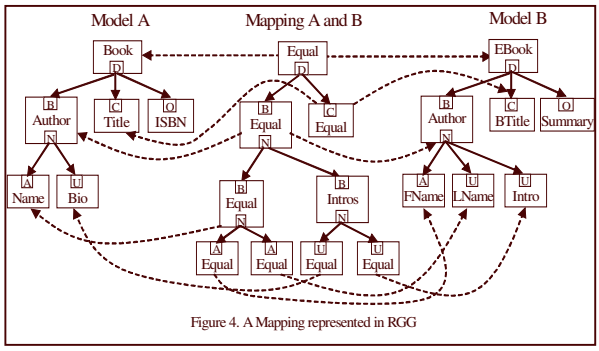

Figure 4. A Mapping represented in RGG

mapping element as vertex F is marked. Productions <3> and <5> define the relationship between the helper and mapping elements, i.e. they may have the has-a relationship in either direction. That a mapping element can have multiple reference elements is specified in Production <4>, and the relation between a mapping element and a reference element is denoted by a dashed edge in this paper.

6. Operators by Graph Transformation

Model management operators take data models and mappings as input and generate another set of data models and mappings as output, and are described by a set of graph transformation rules. This section will go through two operators, Merge and ModelGen, to illustrate the graphical representation of operators. The same principle applies to other operators.

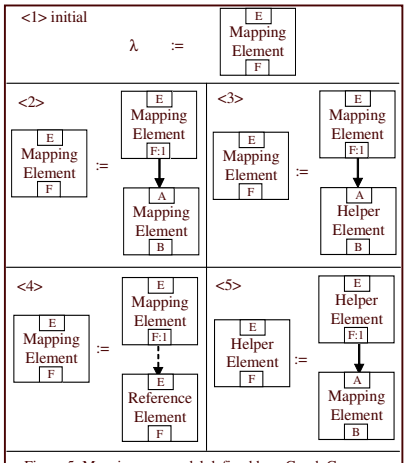

Figure 5. Mapping as a model defined by a Graph Grammar

6.1. Merge Operator

Merge takes three inputs, i.e. model A, model B, and a mapping between A and B, and returns the union model C of A and B along with mappings between C and A, and between C and B [Ber03]. The input of merge is $S = (A, B, M_{AB})$, which consists of three graphs representing model A, model B, and the mapping between A and B. After applying the merge to S, output T consists of five graphs, i.e. $T = (A, B, C, M_1, M_2)$, where A, B are copies of input graphs, C represents the output union model, M_1 and M_2 represent mappings between C and A, and between C and B.

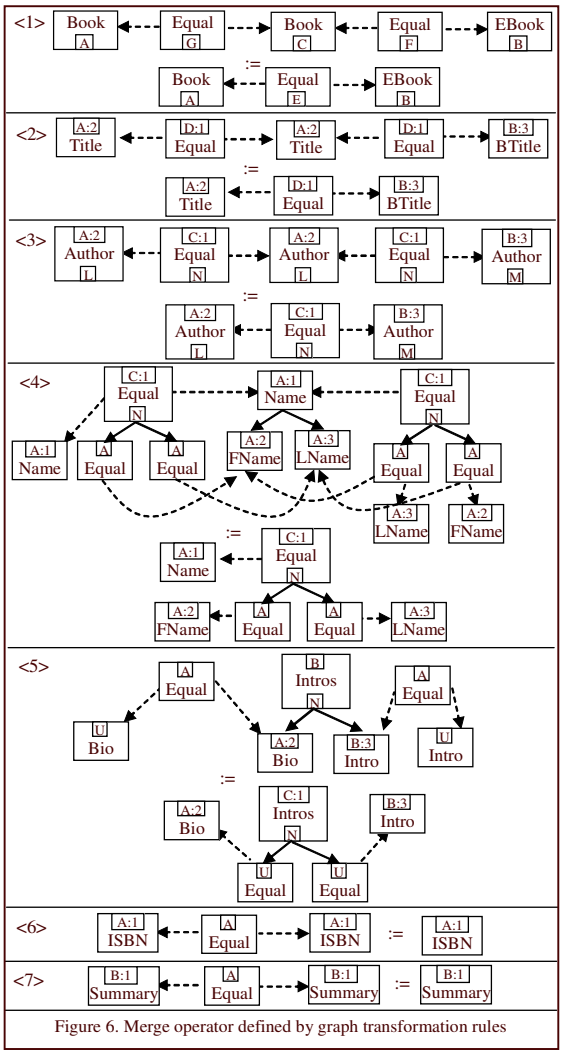

Figure 6. Merge operator defined by graph transformation rules

The semantics of merge can be briefly described as follows: The output of merge is a model that retains all non-duplicated information in A, B, and Map_{AB}; it collapses the redundant information declared by Map_{AB}.

Figure 6 shows a set of graph transformation rules for merging models A and B as defined in Figure 5. Each production rule shows what the result of merge should be. Production <1> defines that root nodes of input models, Book and Ebook, will produce an output data model with a root node Book, and two mappings. Productions <2> and <3> are similar to Production <1>, and copy the referenced node to the output and set a correspondence between the output and input models to form two output mappings. Production <4> merges the structured mappings by defining a new structure in the output model with the nodes referenced by the mapping element and constructing two mappings from elements in the input models to the constructed elements in the output models. Production <5> shows the transformation with a helper element (Intros in this case), and is similar to Production <4>. Productions <6> and <7> copy the input elements that have no reference in the input mapping to the output and establish a mapping between the original element and the copy.

Comparing to an operator algorithm, the graph transformation rules intuitively and explicitly specify what the result should be, and therefore a user with little domain knowledge can manipulate the rules to meet the specific requirements. For example, if one wants to use EBook rather than Book as the root of the output data model, he/she can change the node Book in the left graph of Production <1> to EBook.

6.2. ModelGen Operator

ModelGen takes a model A as input and returns a new model B based on a mapping between A and B [Ber03]. In our framework, ModelGen takes input $S = (A, M_{AB})$, where A is a model, M_{AB} is a mapping, and output is $T = (B)$. ModelGen transforms from input graph S to T by applying a set of transformation rules P, i.e. $T = A\ (S, P)$.

For the input (A, Map_{AB}) in the example of Figure 4, the ModelGen is described by the graph transformation rules in Figure 7. Productions <1>, <2>, and <3> show that the result of a one-to-one correspondence is copied directly from the referenced elements of model B in the mapping. In Productions <4> and <5>, the reference elements in model A are mapped to elements in B via a complex structure of mapping elements or helper elements. For example, Production <4> produces new elements by duplicating reference elements of the mapping, e.g. LName and FName.

The ModelGen on the input (A, M_{AB}) does not produce model B accurately. It cannot produce element summary of the original model B, because the input S (A, M_{AB}) has no such element. To maintain a high fidelity of the output model one can add summary to the left graph, so that the parser will produce the element missing in the output model.

So far two operators, Merge and

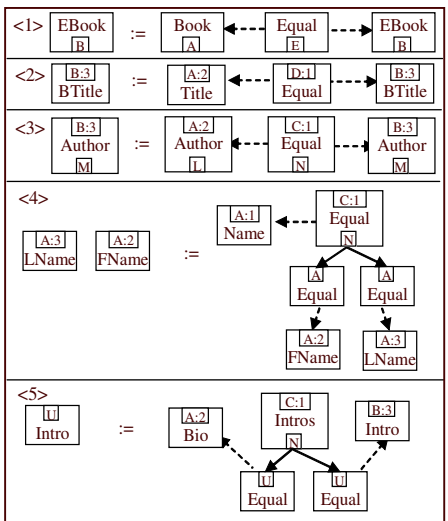

Figure 7. ModelGen by graph transformation rules

ModelGen, are defined by transformation rules on specific inputs. It is easy and feasible for users to specify the specific transformation rules on small-scale inputs, but not for large data models. Therefore the framework automates the process of defining rules for specific inputs by exploiting traditional algorithms, or generalizing the specific graphical operators, as discussed in the following subsection.

6.3. Generalization of Operators

This subsection describes the concept of operator generalization by going through the merge operator. Based on mappings, generalized graph transformation rules visually describe the algorithms for the corresponding operators at a level higher than specific operators. Ideally if we could define all the detailed algorithms of model operators by graph transformation rules, model management could be an automatic and visualized process. Due to the *ad hoc* nature, however, generalized operators still need to be customized for specific inputs, for example the ModelGen in Figure 7 needs to add summary to Production <1> for an accurate output.

Therefore generalization aims at describing algorithms of operators by graph transformation and when applied to a specific input, the parser generates the corresponding specific operators, which are customizable. The framework as shown in Figure 3 could be fully interactive and also visualized.

For example, merge could be generalized as shown in Figure 8, which defines 5 transformation rules. Unlike the merge algorithm, the transformation rules can be customized on the input. Generalized operators do not resolve conflicts, which are to be solved by specific operators. Production <1> merges an elementary mapping, i.e. one to one correspondence as Productions <1>, <2>, and <3> in Figure 6. The output consists of two mappings and one data model together with input elements. In the middle of the left graph of Production <1>, the reference element of the output model is a copy of one of the mapped input elements, the element in model A in this case. The remaining two output reference elements are copies of the corresponding input elements. Two mapping elements on top are output mappings, which map the middle reference element to the left and right reference elements. Production <2>, together with Production <4>, merges the structured mapping elements, such as the equal element of Production <4> in Figure 6. The merge is achieved by making the mapping element and the related reference element a composite element and then extracting the reference element to form the output elements in Production <4>. Similarly Productions <3> and <5> transform the structured helper elements by composing them in <3> and then extracting in <5>.

When the rules are applied to a host graph, the parser will match the nodes in the host graph to the nodes of the same type in the right graph. For example equal in Figure 4 is a mapping element of Figure 8. Because the rules are based on the graph grammar in Figure 6, they can be applied to any host graphs conforming to the grammar.

Similarly, the ModelGen operator in Figure 8 can be generalized and the generic model management visualized. But users cannot customize generalized operators like they do with specific operators. As shown in Figure 3, the two approaches are integrated in our framework, that provides a visual, generic, and customizable model management environment.

7. A Parsing Example

This section describes the transformation process of merging input data models and mapping that were illustrated in Figure 4. The corresponding merge operator is defined in Figure 6. The output includes models A and B (i.e. copies of input), output model C, and mappings Map_{AC} and Map_{BC}.

The first redex found is that of Production <6> in Figure 6, i.e. ISBN of model A. The parser copies ISBN of the model and connects it to the mapping element Equal. A redex of Production <7> is found in the second step, which merges Summary element of model B.

Production <5> is applied in the third step, which

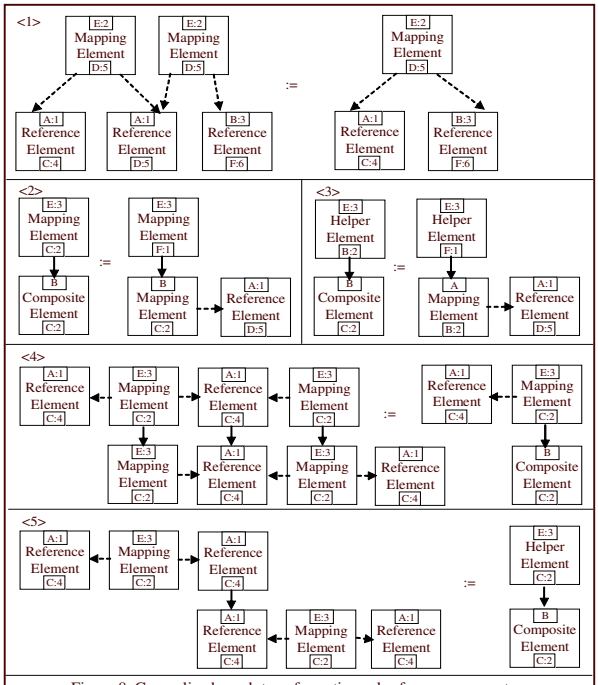

Figure 8. Generalized graph transformation rules for merge operator

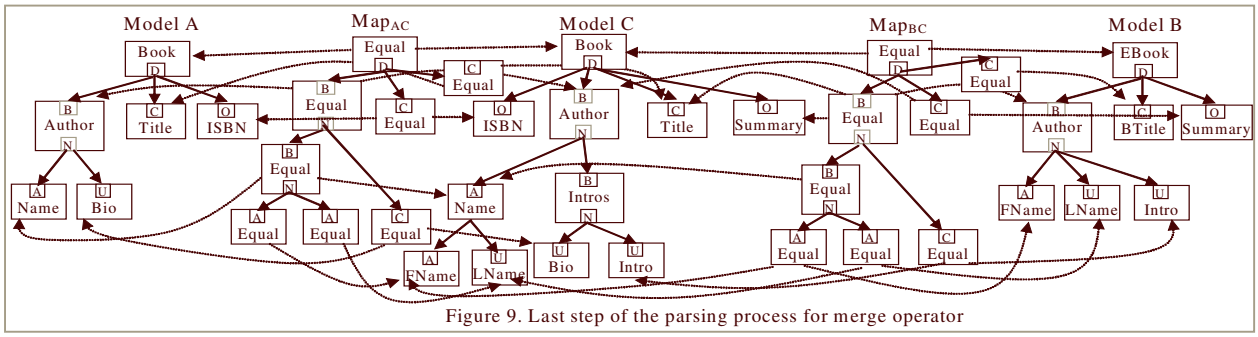

Figure 9. Last step of the parsing process for merge operator

merges mapping with helper elements. The helper element Intros and two connected mapping elements, Bio and Intro, are moved to the output model. Two mapping elements in Map$_{AC}$ and Map$_{BC}$ are connected to Bio and Intro respectively. In the forth and fifth steps, the parser applies Productions <4> and <3> respectively.

Due to the space limit, only the last step is shown in Figure 9, when Production <1> is applied. The mapping between Book and EBook is found as a redex and replaced with two mappings. After the application of this rule, the output model C and two mappings between C and A, B are finally produced.

8. Related Work

Though model management is a relatively new research area, its promising and exciting potential has attracted much attention and made significant advances in several aspects since it was first proposed [Ber00]. In the transformation perspective, according to Bézivin [Béz03], model management may be considered the 3rd generation, with text scripts like the *awk* Unix command being the first generation and tree scripts like XSLT being the second.

Various systems for model management have been presented. Cupid [Mad01, Mad03] and Clio [Mil01] match two models and output the mapping between them, i.e. performing the match operator. Merge has been a hot spot in database research area for a long time. Buneman *et al.* described a theoretical foundation of merge [Bun92]. In the context of generic model management, there are various implementations of the operator, such as Pottinger's approach, which presents the operator based on the BDK algorithm [Pot03], and data integration project Clio [Mil01] that is based on a query language specific to databases or XML schemas. Most of the approaches only concentrate on part of generic model management.

Rondo [Mel03] is the first complete prototype of the generic model management system, in which Melnik *et al.* defined the key conceptual structure of models, mappings, and selectors. They presented an algorithm for the merge operator as an example, and applied it to XML schemas and SQL views. Rondo represents mapping between two data models by a set of correspondences, not as a model as in this paper. Comparing to our interactive and customizable framework, Rondo is like a black box to users and presents no intuitive interface for users to customize.

Model management is also combined with peer-to-peer computing technology [Ber02] and further used as an infrastructure for future Web data representation, notably the semantic Web [Hal03W]. Piazza [Hal03] offers a language for mediating between data sources over the semantic Web. Piazza describes mapping by an adapted query language and has more sophisticated mechanism to retrieve complex data from RDF and XML documents. The appropriate mapping language is derived from XQuery and is complicated for a Web page designer to map some Web pages to others. Users or designers have to solve conflicts manually. The complex query language could potentially hinder the deployment of the Piazza system.

Using graphs to represent and manage data models is not new, and there are many proposals based on graph grammars. Rekers and Schürr presented an ER data model specified by layered graph grammars [Rek97]. Jahnke and Zundorf presented *varlet*, a database reverse engineering environment based on triple graph grammars [Jah98]. The varlet environment supports the analysis of legacy database systems, translation of any relational schema into a conceptual object-oriented schema. More recent work of Wermelinger and Fiadeiro [Wer02] focuses on software architecture reconfiguration using an algebraic approach, i.e. category theory. Consistency of model evolution based on real-time UML is further investigated by Engels *et al.* [Eng02]. These graph transformation based approaches address only specific aspects of model management. No graph-based generic model management system has been proposed.

9. Conclusion

This paper has presented a visual representation of data models and mappings using the RGG, and applied it to model management. Model management operators are specified by graph transformation rules at two levels, i.e. specific operator and generalized operator.

These two operation levels allow experts of data models and graph grammars to define the general rules of operators, and assist users to comprehend and manipulate the specific rules for tuning the result.

The generic, interactive, and visualized model management framework formally defines model management operators and provides intuitive interfaces for users to customize the operators. The graphical representation of operators perfectly matches the interactive nature of model management activities.

Our immediate future work includes the automatic translation of input textual data models to their graphical representations and implementation of operator generalization in the model management framework.

Acknowledgments

We are very grateful to Phil Bernstein for his insightful feedbacks and comments on an earlier version of the paper, which have helped us to improve the paper. The work is partially supported by the National Science Foundation under grant number IIS-0218738.

References

[Ber00] P. A. Bernstein, A. Halevy, and R. A. Pottinger, A Vision for Management of Complex Models, *SIGMOD Record,* 29(4), 2000, 55-63.

[Ber02] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. M., L. Serafini, and I. ihrayeu, Data Management for Peer-to-Peer Computing: A Vision, *Proc. 5th Int. Workshop on the Web and Databases,* Madison, Wisconsin, Jun 2002, 89-94.

[Ber03] P.A. Bernstein, Applying Model Management to Classical Meta Data Problems, *Proc. 2003 CIDR Conf.*, Asilomar, CA, Jan, 2003, 209-220.

[Béz03] J. Bézivin, E. Breton, G. Dupé, P. Valduriez, The ATL Transformation-based Model Management Framework, Research Report No.03.08, Université de Nantes, Sep. 2003.

[Bun92] P. Buneman, S.B. Davidson and A. Kosky, Theoretical Aspects of Schema Merging, *Proc. 3rd Int. Conf. Extending Database Technology,* Vienna, Austria, Mar. 1992, 152-167.

[Eng02] G. Engels, R. Heckel, J.M. Küster, and L. Groenewegen, Consistency-Preserving Model Evolution Through Transformations, *UML'02*, LNCS 2460, Springer, 212-227.

[Ger99] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca, XML-GL: A Graphical Language for Querying and Restructuring XML Documents, *Proc. 8th Int. World Wide Web Conf.*, Toronto, Canada, May 1999, 1171-1187.

[Hal03] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov, Schema Mediation in Peer Data Management Systems, *Proc. Int. Conf. Data Engineering*, ICDE, 2003, 505-518.

[Hal03W] A. Halevy, Z. Ives, I. Tatarinov and P. Mork, Piazza: Data Management Infrastructure for Semantic-Web Applications, *Proc. Int. World Wide Web Conf.,* Budapest, Hungary, May 2003, 556-567.

[Jah98] J. H. Jahnke and A. Zudorf, Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment, Technical Report tr-ri-98-201, University of Paderborn, 1998.

[Mad01] J. Madhavan, P. A. Bernstein, and E. Rahm, Generic Schema Matching Using Cupid, *Proc. 27th VLDB Conf., Roma,* Italy, Sep, 2001, 49-58.

[Mad03] J. Madhavan and A. Y. Halevy, Composing Mappings Among Data Sources, *Proc. 29th VLDB Conf.*, Berlin, German, Sep 2003, 572-583.

[Mel03] S. Melnik, E. Rahm, and P. A. Bernstein, Rondo: A Programming Platform for Generic Model Management, *Proc. SIGMOD 2003 Conf.*, San Dieago, CA, June 2003, 193-204.

[Mil01] R. J. Miller, M. A. Hernandez, L. M. Haas, L. Yan, C. Ho, R. Fagin, and L. Popa, The Clio Project: Managing Heterogeneity, *SIGMOD Record*, 30(1), March 2001, 78-83.

[Pot03] R. A. Pottinger and P. A. Bernstein, Merging Models Based on Given Correspondences, *Proc. 29th VLDB Conf.*, Berlin, Germany, 2003, 826-873.

[Rah01] E. Rahm and P.A. Bernstein, On Matching Schemas Automatically. *MSR Tech. Report MSR-TR-2001-17*, 2001, http://www.research.microsoft.com/pubs.

[Rek97] J. Rekers and A. Schürr, Defining and Parsing Visual Languages with Layered Graph Grammars, *J. Visual Languages and Computing,* 8(1), 1997, 27-55.

[Son04] G.L. Song and K. Zhang, Visual XML Schemas Based on Reserved Graph Grammars, *Proc. Int. Conf. Information Technology: Coding and Computing*, Las Vegas, NV, April 5 -7, 2004, 687- 691.

[Wer02] M. Wermelinger and J.L. Fiadeiro, A Graph Transformation Approach to Software Architecture Reconfiguration, *Science of Computer Programming*, 44, 133-155, 2002.

[Zha97] D.Q. Zhang and K. Zhang, Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs, *Proc. 13th IEEE Symp. Visual Languages,* Capri, Italy, 23-26 Sep. 1997, 284-291.

[Zha01a] K. Zhang, D-Q. Zhang, and Y. Deng, A Visual Approach to XML Document Design and Transformation, *Proc. 2001 IEEE Symp. Human-Centric Computing Languages and Environments*, Stresa, Italy, 5-7 Sep. 2001, 312-319.

[Zha01b] D. Q. Zhang, K. Zhang, and J. Cao, A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages, *The Computer J.*, 44(3), 2001, 186-200.

[Zha01c] K. Zhang, D-Q. Zhang, and J. Cao, Design, Construction, and Application of a Generic Visual Language Generation Environment, *IEEE Trans. Software Engineering*, 27(4), April 2001, 289-307.