Available online at www.sciencedirect.com
ScienceDirect
Journal of Visual Languages and Computing
17 (2006) 508–527

Journal of
Visual Languages
& Computing

www.elsevier.com/locate/jvlc

AutoGen: Easing model management through two levels of abstraction ☆

Guanglei Song[a,*], Jun Kong[b], Kang Zhang[a]

[a]*Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688, USA*
[b]*North Dakota State University, USA*

Abstract

Due to its extensive potential applications, model management has attracted many research interests and gained great progress. To provide easy-to-use interfaces, we have proposed a graph transformation-based model management approach that provides intuitive interfaces for manipulation of graphical data models. The approach consists of two levels of graphical operators: low-level customizable operators and high-level generic operators, both of which consist of a set of graph transformation rules. Users need to program or tune the low-level operators for desirable results. To further improve the ease-of-use of the graphical model management, automatic generation of low level of operators is highly desirable. The paper formalizes specifications of low- and high-level operators and proposes a generator to automatically transform high-level operators into low-level operators upon specific input data models. Based on graph transformation theoretical foundation, we design an algorithm for the generator to automatically produce low-level operators from input data models and mappings according to a high-level operator. The generator, called AutoGen, therefore eliminates many tedious specifications and thus eases the use of the graphical model management system.

Keywords: Model management; Graph transformation; Graph grammar; Visual programming; Schema interoperation

☆The work is partially supported by the National Science Foundation under Grant no. IIS-0218738.
*Corresponding author.
E-mail addresses: gxs017800@utdallas.edu (G. Song), jun.kong@ndsu.edu (J. Kong), kzhang@utdallas.edu (K. Zhang).

1045-926X/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2006.10.004

1. Introduction

With the advance of Internet applications, interoperation among different formats is becoming critically important. Many approaches have been intensely researched to provide systematic solutions for manipulating heterogeneous data sources, such as peer-to-peer data management [1]. Manipulating enormous heterogeneous schemas, however, has been relatively a forgotten research area. These heterogeneous schemas, such as XML Schemas [2], RELAX [3], SOX [4], ER models, SQL schemas and so on, are used to define data sources, called *meta-data* or *data models*. Traditional approaches to manipulating these data models are manually specified or designed case by case for specific domains, i.e. object-at-a-time. Information engineers have to put much effort to program specifically for the data model applications concerning data migration, data integration and translation. Such processes are time-consuming and error-prone, and eliminate the possibility of reuse.

To reduce the programming effort, Model Management [5] is introduced to reconcile the painful process of manipulating heterogeneous data models. According to the vision paper [5], model-related applications can be composed by a sequence of atomic operations on data models, such as Merge, Match and so on. These atomic operations are defined as generic operators such that they treat data models as high-level data structures and therefore can be re-used in various domains. A model management system provides a set of high-level programming interfaces for applications to implement the atomic operations to save programming effort. Model management is the first effort to organize and generalize these operators to a systematic architecture. Conceptually, these model management operators have been applied to solve many classic meta-data problems successfully [6] and the first textural prototype system has been developed [7].

Given the operators provided by a model management system, users need to write a program to combine a series of operators to fulfill a specific task. Each execution process of a specific operator is transparent to the user and not customizable. Many usage scenarios, such as the motivating example of Melnik et al. [8], however have demonstrated that user interventions are constantly required and customizability is highly desirable for model management operators. Existing operators, however, are defined by text and transparent to users, and their implementations are hard-coded in the system. Little work has been done to improve the customizability and user interfaces of model management operators.

To improve the expressiveness and customizability of model management operators, we recently proposed a graphical model management framework [9] based on a graph grammar formalism, i.e. the Reserved Graph Grammar [10,11]. We also presented graphical definitions and representations of data models and mappings. Many data models, including ER models and UML models, are represented by graphs and others, including XML Schemas and SQL schemas, can easily be translated into graphs [7]. With intuitive representations for designers to communicate with each other, graphs are natural representations for data models. Graph transformation, as the theoretical foundation of visual programming languages, is capable of formally defining how graphs should be built and how they evolve [9]. The framework defines operations on data models through a set of graph transformation rules. These transformation rules are declarative and customizable.

The framework provides two levels of graphical operators, i.e. *low level* for end-to users or adjust and *high-level* operators for domain-experts to program. The two tier architecture

provides interfaces for both domain experts and end-users for their distinct requirements. A low-level operator consists of a set of graph transformation rules that defines operations on specific data models. Through low-level operator, users can customize and tune the low-level operator for desired results. A high-level operator consists of a set of generic rules, and can be applied to a set of data models. Domain experts program such high-level operators and provide them to end-users for generic use in a domain. The two levels of graphical operators are defined in two different abstraction levels of information. Low-level operators describe the operation specific to data model instances, e.g. an XML schema, while high-level operators define the general guidance for operations among a set of data models, such as RELAX schemas.

Manually defining a low-level operator, which consists of a set of specific graph transformation rules, from the scratch is time-consuming. The syntax-directed guidance for drawing the specific rules is provided by the framework to ease the process, while the framework however has no automatic mechanism to ease the programming process through the translation from a high-level operator to a low-level operator.

This paper presents formal specifications for high- and low-level operators. High-level operators are defined by a set of *generic rules* and low-level operators are defined by a set of *specific rules*. Based on the formal specifications, this paper proposes an automatic generation mechanism for generating low-level operators to save the programming effort. To realize the automatic generation process, we present a generator that transforms a high-level operator (generic rules) into a low-level operator (specific rules) according to input data models so that the user's effort in programming specific rules is minimized. The paper makes the following contributions to the research community:

- formal specifications for high- and low-level operators;
- formal definition of a generator for generating specific rules; and
- an automatic generation algorithm for the generator.

The remaining of the paper is organized as follows: Section 2 introduces the Reserved Graph Grammar formalism. Section 3 presents the graphical model management, including graphical data models, mappings and operators on them. Section 4 presents the formal specifications for the generator and the automatic generation mechanism, followed by the two illustrative examples in Section 5. Section 6 compares the work with related works and Section 7 concludes the paper.

2. The reserved graph grammar formalism

As a context-sensitive grammar, the RGG formalism is powerful in expressing various types of diagrams, with a parsing complexity of polynomial time under a non-ambiguous condition [10–12]. It is expressed in a node-edge format, similar to a "box and line" drawing [13] but devised to suit automatic analysis through graph reasoning. In an RGG, nodes are organized into a two-level hierarchy, where a large rectangle representing the node itself is the first level with embedded small rectangles as the second level called *vertices*. For example, inside the left dashed box of Fig. 1, a node Item has two vertices, i.e. P and C. In a node, each vertex is uniquely identified. The name of a node distinguishes the node's type, similar to the type of variables in conventional programming languages. A node can be viewed as a module, a procedure or a variable, etc., depending on the

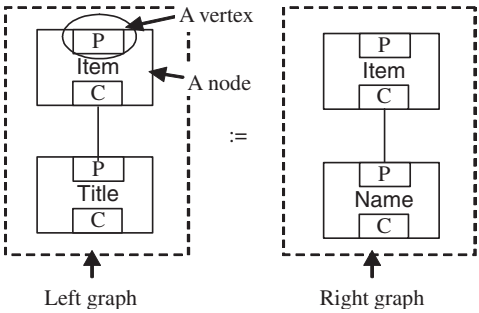

Fig. 1. An example rule.

domain requirement and granularity. Edges are used to denote relationships between nodes. A vertex functions as the point attached to an edge. Edges are used to denote communications or relationships between nodes [14].

Based on nodes and edges, the RGG offers a formal approach to specifying the evolution of graphs, called *host graphs*. In general, a graph grammar consists of a set of graph rewriting rules, each having two graphs that are called *left graph* and *right graph*. A simple RGG rule is shown in Fig. 1, where a left graph is inside the left dashed box and a right graph is inside the right dashed box.

In the RGG, the *application* of a rule to a host graph, i.e. a graph transformation, replaces a sub-graph in the host graph that matches the right graph of the rule by the left graph [11]. The rewriting rule shown in Fig. 1 describes the transformation from the node Name to Title. The RGG uses the *marking* technique, which classifies vertices as marked and un-marked ones, to address the embedding issue, i.e. building connections between the replacing sub-graph and the surrounding of the replaced sub-graph in the host graph. A marked vertex is identified by a unique integer, and preserves its associated edges connected to nodes outside a replaced sub-graph.

The RGG is equipped with a deterministic parsing algorithm, called selection-free parsing algorithm (SFPA) [10]. A graph grammar must satisfy the selection-free condition in order to use SFPA. Informally, the selection-free property ensures that different orders of applications of rewriting rules result in the same result. We developed an algorithm to automatically check whether a graph grammar satisfies the selection-free condition [15]. Though it is unclear how this condition limits the application scope, it is interesting to note that even grammars for some complicated graphs satisfy the condition [16]. We proved that a failed parsing path indicates an invalid graph, and thus SFPA is efficient with a polynomial parsing complexity by only trying one parsing path [15].

3. Model management by graph grammars

A model management environment offers operators that generalize the transformation operations for various metadata applications as follows [6]:

- 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 a 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. This paper focuses only on Merge operator and the same principle can be applied to ModelGen, Diff and Compose. Match [17] is however out of the scope of this paper and intense research has been done recently. Each of these operators is a major topic in the research community [5] and the visual operators in this paper present improvement due to full utilization of graphical presentations.

Graphs offer intuitive means for describing data models. Operators on data models define the conceptual transformations of input data models and can therefore be defined by graph transformation rules. The graphical representation of operators and their transformation are intuitive, and ideally match the interactive nature of model management operations.

3.1. Graphical data models

Similar to ER models, we represent a data model by a node–edge diagram, where a node represents an object, element or entity and an edge represents the relationship between two nodes.

A data model, or a schema, is represented by a graph G, which consists of a set of nodes N, and a set of edges E: $N \times N$. Each node or edge may have a name, which composes a set of labels L. The graph for a data model can be defined as the following:

Definition 1. A host graph of a data model is $G = (N, E, L, F)$, where N is the node set, E is the edge set: $N \times N$ and F is a function F: $N \cup E \rightarrow L$, L is a set of labels.

A host graph represents a simple data model, e.g. an XML schema. As shown in Fig. 2, the graph defines an element Lists with three child elements, e.g. Title, Bids, and Price, and the element Bids has two child elements, Date and Amount. The XML schema is instantiated in a set of XML files encoding online bid items of different names and prices. Every node in Fig. 2 represents an element with vertex P linking to the parents and C linking to the children and K linking to the attributes or keys.

We use a graph grammar to define the syntax of such graphs. As shown in Fig. 3, a graph grammar defines the syntax of graphs for the XML schema, which is simplified to fit in the paper as an illustration. The graph grammar consists of 4 rules, and the ith rule is marked with $\langle i \rangle$. Rule $\langle 1 \rangle$ specifies that a schema consists of at least one root Element. Rule $\langle 2 \rangle$ defines that Element can have any number of child elements. Rule $\langle 3 \rangle$ defines that each Element can have many attributes. Rule $\langle 4 \rangle$ specifies each Element can have a key.

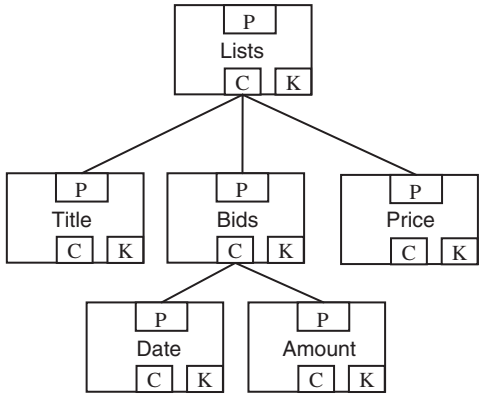

Fig. 2. A data model by a graph.

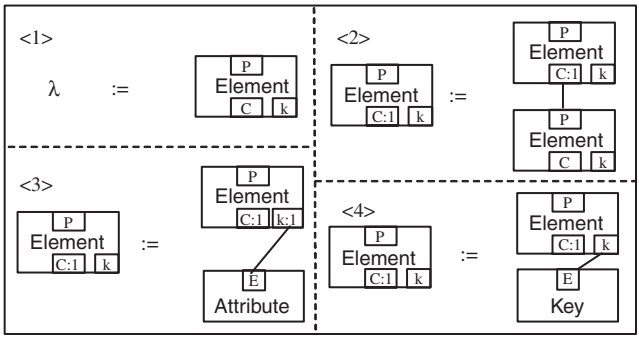

Fig. 3. Some rules for XML schemas.

3.2. Graphical mappings

A mapping, Map_{AB}, defines how models A and B are related [18]. Based on the graphical representation of data models, the graph is a straightforward choice for the representation of mappings. Many proposals use graphical metaphors to represent schema mappings like in Rondo [7], and Clio [19]. These proposals represent a mapping as a set of lines connecting the elements of two data models. Such a representation is simple but not as powerful as SQL view [20] or as a data model [6]. SQL view can represent powerful semantic similarity of elements in two data models, but the SQL view is not generic and cannot represent mappings among heterogeneous data sources other than relational databases, such as XML schemas. On the other hand, mappings are structural instead of flat bi-directional, and thus cannot be described by those simple two-way correspondences. The mapping structure described by Bernstein et al. [5] is an appropriate compromise, and is generic yet powerful.

Similarly, we represent mappings as special data models. As shown in Fig. 4, a mapping itself is represented as a data model. A graph representing a mapping is called a *mapping graph*. Each mapping graph consists of edges and nodes, representing relationships and

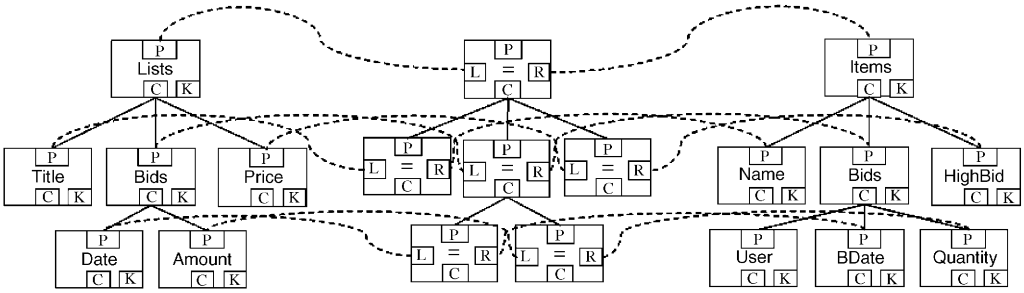

Fig. 4. A mapping graph.

elements, respectively. A mapping graph has two relationship types, i.e. has-a and refer-to, and two element types, i.e. *mapping* and *referral*. A *mapping element* specifies how the two referenced models' elements are related, such as equality using a node " = " in Fig. 4. A referral element is a reference to the element of the two corresponding models, such as Lists in Fig. 4. A dashed line denotes the refer-to relationship between a mapping element and a referral element.

Solid lines in Fig. 4 between two referral elements, for example the link between nodes Lists and Title, are defined by corresponding model graphs rather than the mapping graph. The mapping graph in Fig. 4 shows these links only to illustrate structure of data models and the links are not part of the mapping graph.

Since a mapping is treated as a special case of data models, which are represented by graphs, the syntax of mappings is also defined by a graph grammar as for a regular data model. The graph grammar for mapping graphs is defined in our previous work [9].

So far both types of inputs, i.e. data models and mappings, are represented by graphs, which are further defined by a graph grammar. Given the fact that outputs of operations on them are also data models and mappings, operations on them are considered as graph transformations from input graphs to output graphs as described in the next section.

3.3. Graphical operators

Each model management operator has specific semantics for how to manipulate the inputs, e.g. data models and mappings. Since the inputs and outputs of the operators are all represented by graphs, it is straightforward to describe an operator by a set of graph transformation rules. The RGG has been enhanced to easily define graph transformations [16], and then adapted to represent the model management operators [9].

We take merge as an example operator to illustrate the transformation process. Merge has been a hot research topics for many years [18] and complex enough to verify the idea. The input of operator 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 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 model, M_1 and M_2 represent mappings between C and A, and between C and B, respectively. The output data model C retains all

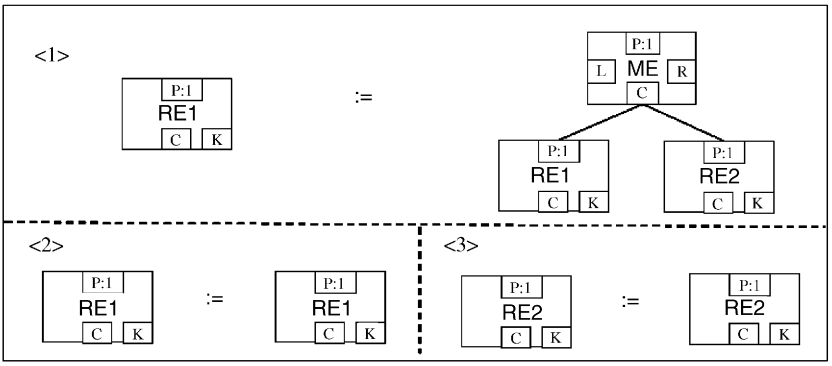

Fig. 5. Generic rules for merge.

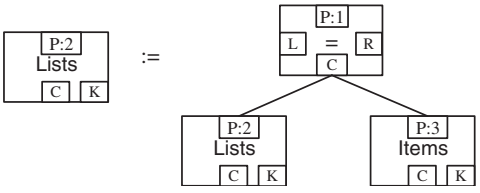

Fig. 6. A specific rule for merge.

non-duplicated information in models A and B. It collapses the redundant information declared by input mapping M_{AB}.

The semantics of the merge operator can be generally defined by a set of graph transformation rules. We consider the rules to be generic, since they can be applied to different data models but cannot be easily customized.

For the mapping described in Fig. 4, we can define a set of rules as shown in Fig. 5. ME is a mapping element, and RE is a referral element. Subscripts 1 or 2 denotes the origin of the element, for example, RE_1 represents an element from Model 1. Concentrating on input and output data models, the rules simplify the operator by removing some transformations, such as output mappings, which specify the mapping relationship between input elements and output elements. Since the output elements are copied from input elements, the mapping between them can be easily replicated from the copying process. Rule $\langle 1 \rangle$ shows that two mapped elements RE_1 and RE_2 can be merged into one element RE_1. Rules $\langle 2 \rangle$ and $\langle 3 \rangle$ define that an unmapped element from inputs 1 or 2 should be copied to the output.

Comparing to an algorithm for the same operator, the graph transformation rules intuitively and declaratively specify results, and therefore a user with little domain knowledge can manipulate the rules to meet specific requirements. But the generic rules are not tailored for the input data models and mappings, and hard to customize.

Therefore the concept of specific rules is introduced as shown in Fig. 6 for users to customize the result by tuning the rules rather than results. The specific rule in Fig. 6 defines that Lists and Items are mapped together via the mapping element " = ", and should be merged together to be a single node Lists in the output graph. A user can simply

change the concrete correspondence between Lists and Items nodes to adjust the result. The change made to the specific rule does not change other elements and mappings because the rule is specific to the nodes. For example, if one wants to use Lists rather than Items as an element of the merged data model, he/she could simply change the node Lists in the left graph of a production to Items.

After applying the merge operator of either generic rules or specific rules, the input model graph will be translated into an output model graph.

4. A generator for specific rules

As described in Section 3, both generic and specific rules play important rules in visual model management and present the two levels of operators for different users. This section presents formal specifications for the operators and introduces a generator for automatic generation of specific rules from generic rules.

4.1. Automatic generation overview

Generic rules are at a high level and can be applied to many input data models and mappings, while specific rules are customizable. To take advantages of both types of rules, we propose a mechanism for automatic generation of specific rules from generic rules. The idea is to automatically generate a set of rules specific to the input data models with detailed names and relationships based on the same transformation principle as that of generic rules. The generation process will be covered in more detail in Section 4.4.

As shown in Fig. 7, upon inputs the two types of transformation rules are executed through two types of transformers, respectively, and produce results. The two types of transformers will be discussed in Section 4.3.

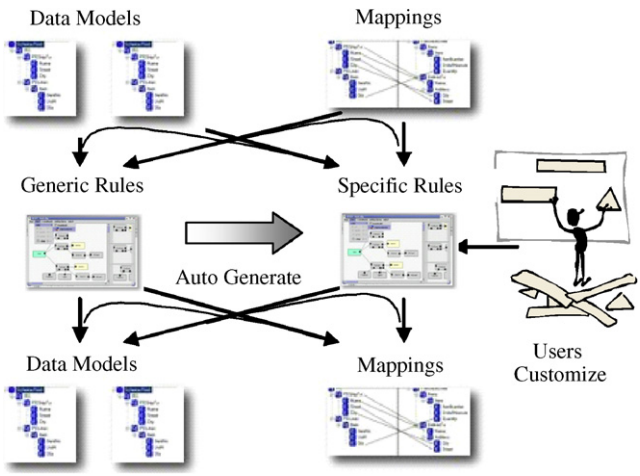

Fig. 7. Overview of automatic generation process.

The set of generic rules for an operator, such as the example in Fig. 5, defines the principle for manipulating input data models and mappings between them, i.e. collapse mapped elements into one output element and copy un-mapped elements for the merge operator example in Fig. 5. Through a transformer, generic rules can be applied to specific input data models directly.

The specific rules for an operator, which is specific to the inputs, are also used by a transformer to translate input data models and mappings to another set of data models and mappings. As shown in Fig. 7, contrary to generic rules, specific rules are customizable and users are given opportunities to tune the results.

Both types of transformation rules are necessary for users to manipulate data models and mappings and construct model-related applications. The original design however lacks a proper transition from generic rules to specific rules in case the user demands a set of new specific rules to be generated from existing generic rules. As shown in the middle of Fig. 7, the automatic generation of specific rules bridges the gap between specific rules and generic rules. Upon input data models and mappings, a generator, to be discussed in Section 4.3, is proposed to automatically generate specific rules according to generic rules.

With the automatic generation mechanism, the two level graphical operators can be used in a typical scenario: the graphical generic rules are designed and tested by model management and domain experts and end-users can simply apply the generic rules to specific inputs and adjust results via automatically generated specific rules. The automatic generation mechanism eases the usage of the model management system.

4.2. Preliminaries

As defined in Section 3, a host graph, i.e. graphical representation of a data model or mapping, is denoted by $G\,(E, N, L, F)$, where $G\,E$ denotes the edge set E of a graph G, $G\,N$ denotes the node set of a graph G, the label set used for the graph G is called L and $G\,F$ denotes the function $E \cup N \to L$. Each node in the graph is defined by $n = (s, V, l)$, where $s \in N$ is a node, and l: $V \to L$ defines labels of vertices. $n\,V$ denotes all the vertices of node n.

Two nodes n_1 and n_2 are *isomorphic* to each other, denoted by $n_1 \approx n_2$, iff they have the same vertex labels (including nodes) [10], i.e.

$$\exists f((f : n_1 V \to n_2 V) \wedge \forall v \in n_1 V(n_1 l(v) = n_2 l(f(v))) \wedge n_2 s = f(n_1 s)).$$

The definition indicates that two nodes with the same vertex and node labels are isomorphic to each other. Two graphs G_1 and G_2 are isomorphic, iff $\exists f$: $G_1 \to G_2$, where f is a mapping such that

$$\forall n \in G_1 N : n \approx f(n) \text{ and } \forall e = (v_1, v_2) \in G_1 E : f(e) = (f(v_1), f(v_2)) \in G_2 E.$$

The *redex* is a sub-graph of a host graph that is isomorphic to the right graph. Two graphs are isomorphic if they have the same structure and nodes have the same labels.

The parsing algorithm that matches two nodes with the same labels, i.e. *isomorphism*, is called *label-based parsing algorithm*.

Specific rules explicitly define the transformation based on node labels so that users are able to customize. The label-based parsing algorithm is employed to parse specific rules. The parsing algorithm searches in the host graph for a redex of the right graph of a specific

rule by comparing labels of nodes rather than types and replaces the redex by the left graph. Node types are used by the parsing algorithm for generic rules and will be discussed in detail in Section 4.3. Therefore, the parsing algorithm does not match the node " = " in a host graph with the node ME in the right graph due to distinct labels of the two nodes. The parsing algorithm searches and replaces to produce output data models and mappings until no more redex is found.

4.3. Generator definitions

When a generic rule is applied to a host graph, the transformer needs to match a node in the host graph with a node of the same type in the right graph. For example " = " in Fig. 4 is of ME type in Fig. 5, and the transformer needs to match the node " = " with ME and perform transformation according to the generic rule. As discussed in Section 4.2, the label-based parsing algorithm is not applicable to the translation against generic rules. A new transformer that translates against generic rules is needed. This section also introduces a generator that outputs a set of specific rules, rather than a new data model graph, upon input models according to generic rules.

Generic rules define the transformation based on the syntax definition of data models rather than a specific data model, therefore a match between a subgraph of the host graph and a right graph can only be found via node types that have one-to-many correspondences. We add a type to each node in the generic rules and a type definition for each node of a host graph besides the label.

A host graph is therefore defined as $G' = (V, E, C, L, F, M)$, where C is a set of node types and M: $V \rightarrow C$ is a function defining the type of each node. With this extension, a node in the host graph for the automatic generation of specific rules is defined as follows:

Definition 2. $n = (s, c, V, l)$ is a node on the label set L, where V is a set of vertices, s is the node itself and c is the type of the node and l: $V \rightarrow L$ is an injective function of mapping from V to L.

The node n_1 is homomorphic to node n_2, called $n_1 \approx n_2$, if they have the same vertex labels, i.e. $n_1 V = n_2 V$ and they have the same type, i.e. $n_1 c = n_2 c$.

Definition 3. Two graphs G_1 and G_2 are homomorphism, iff $\exists f$: $G_1 \rightarrow G_2$ is a mapping such that

$$\forall n \in G_1 N : n \approx f(n) \text{ and } \forall e = (v_1, v_2) \in G_1 E : f(e) = (f(v_1), f(v_2)) \in G_2 E.$$

The parsing algorithm that searches for a redex by finding a homomorphism of the right graph of a rule is *type-based parsing algorithm*, which is used to parse and translate a host graph against generic rules.

For a generic rule, a redex is a subgraph of the host graph that is homomorphic to the right graph of the rule. During the parsing process, the parsing algorithm for generic rules searches for a homomorphism of the right graph of a generic rule in the input host graph. A homomorphism is a sub-graph of the host graph, i.e. type-based redex. The parsing algorithm replaces the redex with the left graph and continues to search for the next redex until no more redex exists.

Fig. 8 illustrates a redex of rule 1 in Fig. 5 against the host graph in Fig. 4. The node " = " in the host graph is homomorphic to node ME of the right graph, i.e. " = " $\approx$ ME,

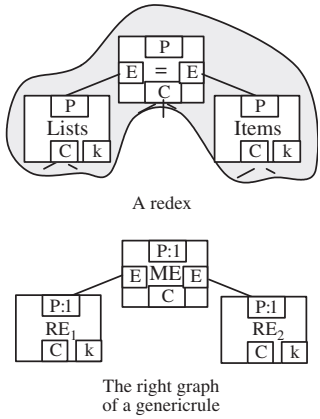

Fig. 8. A redex of the right graph of a generic rule.

because the type of the node " = " is ME and has the same vertices as those in ME. Similar principle applies, RE_1 is homomorphic to Items and RE_2 is homomorphic to Items. The subgraph (" = ", Items, Lists) has the same edges as those of the right graph. According to the Definition 3, the subgraph (" = ", Items, Lists) is homomorphic to the right graph of Rule $\langle 1 \rangle$ and therefore is a redex for the rule.

A high- or low-level operator, e.g. Merge or ModelGen, is defined by a set of rules as follows: $OP = R$, where R is a set of transformation rules such that $\forall r \in R : r = (LG, RG)$, where LG and RG are left and right graphs.

The input to a model management operator consists of one or more data models and mappings, each of which is represented by a graph. The transformer parses a single host graph rather than a set of graphs against transformation rules and the input graphs need to be converted into a single host graph before being parsed by the transformer. Regarding all model management operators, possible combinations of input model graphs and mapping graphs include two data models only, one mapping with one or two data models, and two mappings. If the input consists of two data models, the conversion process simply copies model graphs into one host graph. In this case, the conversion combines a mapping graph with the corresponding model graphs. With two mappings, a host graph is produced from the union of two input mappings in the same fashion as converting two data models. The conversion process is defined as follows:

Given two model graphs or mapping graphs G_1 $(V_1, E_1, C_1, L_1, F_1, M_1)$ and G_2 $(V_2, E_2, C_2, L_2, F_2, M_2)$, their union $G = G_1 \cup G_2$ is defined as: $G = (V_1 \cup V_2,\ E_1 \cup E_2,\ C_1 \cup C_2,\ L_1 \cup L_2,\ F,\ M)$, where F and M are new functions such that F: $V_1 \cup V_2 \cup E_1 \cup E_2 \rightarrow L_1 \cup L_2$ and M: $V_1 \cup V_2 \rightarrow C_1 \cup C_2$.

Further, the new functions F and M are defined as

$$F(x) = \begin{cases} F_1(x) \ldots x \in V_1 \cup E_1, \\ F_2(x) \ldots x \in V_2 \cup E_2, \end{cases} \text{and } M(x) = \begin{cases} M_1(x) \ldots x \in V_1, \\ M_2(x) \ldots x \in V_2. \end{cases}$$

Similarly, a mapping graph G_m can be combined with two corresponding data models G_1 and G_2 based on referral elements of G_m to elements related in G_1 and G_2.

Given a mapping graph G_m ($V_m, E_m, C_m, L_m, F_m, M_m$) and its corresponding data model graphs G_1 ($V_1, E_1, C_1, L_1, F_1, M_1$) and G_2 ($V_2, E_2, C_2, L_2, F_2, M_2$), their union G is defined as: $G = (V, E, C, L, F, M)$, where $V = V_1 \cup V_2 \cup V_m$, $E = E_1 \cup E_2 \cup E_m$, $C = C_1 \cup C_2 \cup C_m$, $L = L_1 \cup L_2 \cup L_m$, and F and M are new functions such that F: $V_1 \cup V_2 \cup V_m \cup E_1 \cup E_2 \cup E_m \rightarrow L_1 \cup L_2 \cup L_m$ and M: $V_1 \cup V_2 \cup V_m \rightarrow C_1 \cup C_2 \cup C_m$. Similar to the union of model graphs, the union of vertices of a mapping graph and a model graph is also regarded as a regular union except that a referral element of a mapping graph is considered as the same node of the corresponding element in model graph.

Further, the new functions F and M are defined as

$$F(x) = \begin{cases} F_1(x) \ldots x \in V_1 \cup E_1, \\ F_2(x) \ldots x \in V_2 \cup E_2, \\ F_m(x) \ldots x \in V_m \cup E_m, \end{cases} \quad \text{and } M(x) = \begin{cases} M_1(x) \ldots x \in V_1, \\ M_1(x) \ldots x \in V_1, \\ M_m(x) \ldots x \in V_m. \end{cases}$$

By the union operations, input data model graphs and mapping graphs are preprocessed to become a single unconnected graph, which is regarded as the input host graph to a transformer.

A generator is introduced to reconcile the transition from generic rules to specific rules and ease the effort in programming a set of specific rules. We call the generator *AutoGen*, which is a new atomic operator in visual model management system.

> AutoGen takes data models and/or mappings as an input graph G and a set of generic rules GR and produces specific rules SR for the corresponding generic rules, i.e. $SR = \text{AutoGen}(G, GR)$.

Depending on the type of operator defined by the generic rules, the AutoGen requires necessary input data models or mappings correspondingly. For example, an input data model and a mapping are required for the generic rules of ModelGen operator, while AutoGen requires two input data models and a mapping for merge. AutoGen employs a generation algorithm to parse the inputs against generic rules. The generation algorithm is described in details in the Section 4.4.

4.4. A generation algorithm

We base the generation algorithm on the SFPA for the RGG [11]. The generation algorithm has two modules, i.e. redex searching and redex application, as illustrated in Fig. 9. It is a sequence of applications, which is modeled as recognize-select-execute, and proceeds as follows:

1. make a copy of the host graph;
2. search in the host graph for a redex of the right graph of a generic rule;
3. replace the redex by the left graph of the rule and generate a new specific rule; and
4. repeat steps 2 and 3 until the host graph is empty or no redex is found.

Particularly, in search of a redex in the host graph for a generic rule, the algorithm employs the FindRedex algorithm as shown in Fig. 10 to find the redex. The algorithm searches in the host graph to match a node in the right graph of a generic rule. Once a node

```
  SpecificRuleSet Generation (HostGraph host, GenericRule GR)
  {
1:      outputgraph = host;
2:      SpecificRuleSet SRS;
3:      while host graph is not empty do
4:      {
5:            matched = false;
6:            for all r ∈ GR
7:             {
8:                      Redex rex = FindRedex (host, r);
9:                      If (rex is not empty)
10:                      {
11:                        SpecificRule SR = ApplyRedex(host, outputgraph, r, rex);
12:                        SRS.add(SR);
13:                        Matched = true;
14:                     }
15:            }
16:            if (matched == false)
17:            {
18:                  print ("Invalid");
19:                  exit(0);
20:            }
21:     }
22:     Return SRS;
  }
```

Fig. 9. The AutoGen algorithm.

```
  FindRedex (HostGraph host, Rule r)
  {
1:      rightGraph = r.rightGraph;
2:      Init redex;
3:      while(true)
4:      {
5:         n1 = searchNode(host, rightGraph);
6:         If (n1!=null)
7:         {
8:               e1= searchEdge(host, rightGraph, n1);
9:               if (e1!=null)
10:              {
11:                    redex.addNode(n1);
12:                    redex.addEdge(e1);
13:              } else  redex.delete(n1);
14:        } else return null;
15:        if(redex.equal(rightGraph))
16:            break;
17:     }
18:     return redex;
   }
```

Fig. 10. Finding a redex.

is identified, the algorithm proceeds to all the neighboring nodes until the subgraph is fully matched or no more nodes can be matched. In the former case, a redex of the current rule is returned. In the latter case, the algorithm returns no redex.

Once the algorithm finds a redex, an application of the rule is performed by the ApplyRedex algorithm shown in Fig. 11. Lines 1 and 2 delete the redex and embed the left graph of the rule into a copy of the host graph. Line 3 deletes the replaced part of the subgraph in the original host graph to validate the input host graph against generic rules.

```
  SpecificRule ApplyRedex (HostGraph host, HostGraph outputGraph,
                  Rule r, Redex rex)
  {
1:      outputGraph.delete(rex),
2:          and host.insert(r.leftGraph);
3:      Graph sub = host.delete(rex);
4:
5:      New r1;
6:      r1.createRightGraph(rex);
7:      r1.createLeftGraph(sub);
8:      return r1;
  }
```

Fig. 11. Creating a specific rule.

After embedding the left graph, the algorithm creates a new specific rule, which encloses the embedded subgraph as the left graph and the redex as the right graph.

After applying generic rules to a host graph, the algorithm produces a set of specific rules. The automatic generation mechanism bridges the gap between generic rules and specific rules, and presents users a set of customizable rules towards the data model (the output specific rules). After users' adaptation, the specific rules may become inconsistent and cannot be applied to any input data models or mappings. With the specific rule's customizability, users may adjust the specific rules to produce desirable results.

5. Illustrative examples

This section presents two examples to illustrate how AutoGen generates specific rules for two operators Merge and ModelGen.

5.1. A merge example

When AutoGen takes generic rules of the merge operator as the input rules, the input host graph contains two model graphs and their mapping graph.

The mapping graph in Fig. 4 describes a mapping between two data models. It is produced by combining the two corresponding model graphs and the mapping graph as described in Section 4.3. Therefore this mapping graph is the input host graph to AutoGen for the merge operator. Upon its input, AutoGen produces a set of specific rules as described in Section 4 according to generic rules.

The generic rules of a merge operator illustrated in Fig. 5 are considered the other input to AutoGen. After parsing the input host graph according to the generic rules, AutoGen produces a set of specific rules shown in Fig. 12. Each of the 7 specific transformation rules describes a mapping as the right graph and an output graph as the left graph.

The user can easily customize these rules for distinct purposes. For example the user can rename the left graph of rule $\langle 1 \rangle$ to Lists so that the specific rules will produce output data model with a root named Items rather than Lists.

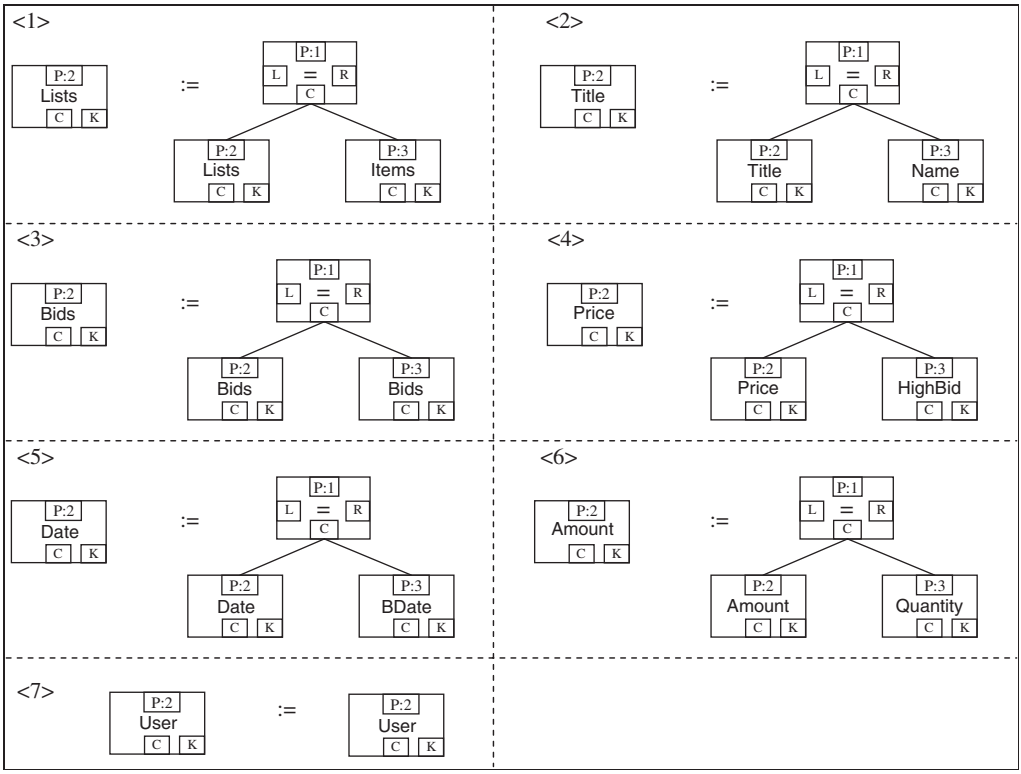

Fig. 12. Specific rules for the Merge operator. generated by AutoGen upon the input mapping graph in Fig. 4.

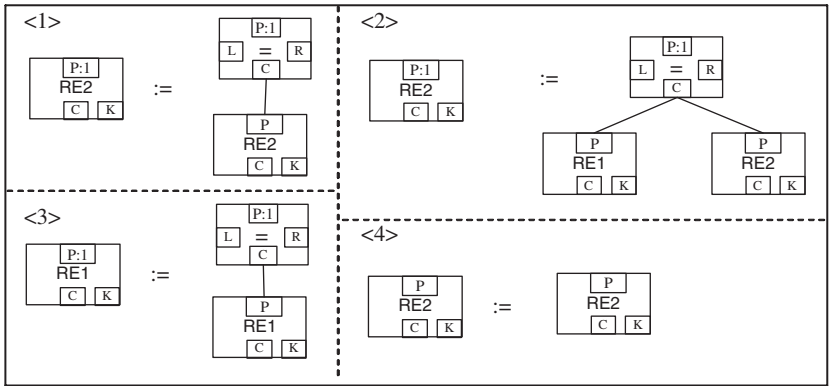

Fig. 13. Generic rules for the ModelGen operator.

5.2. A ModelGen example

When AutoGen takes generic rules for the ModelGen operator as the input rules, the input host graph contains a model graph and a mapping graph. AutoGen generates a set of specific rules for the ModelGen operator.

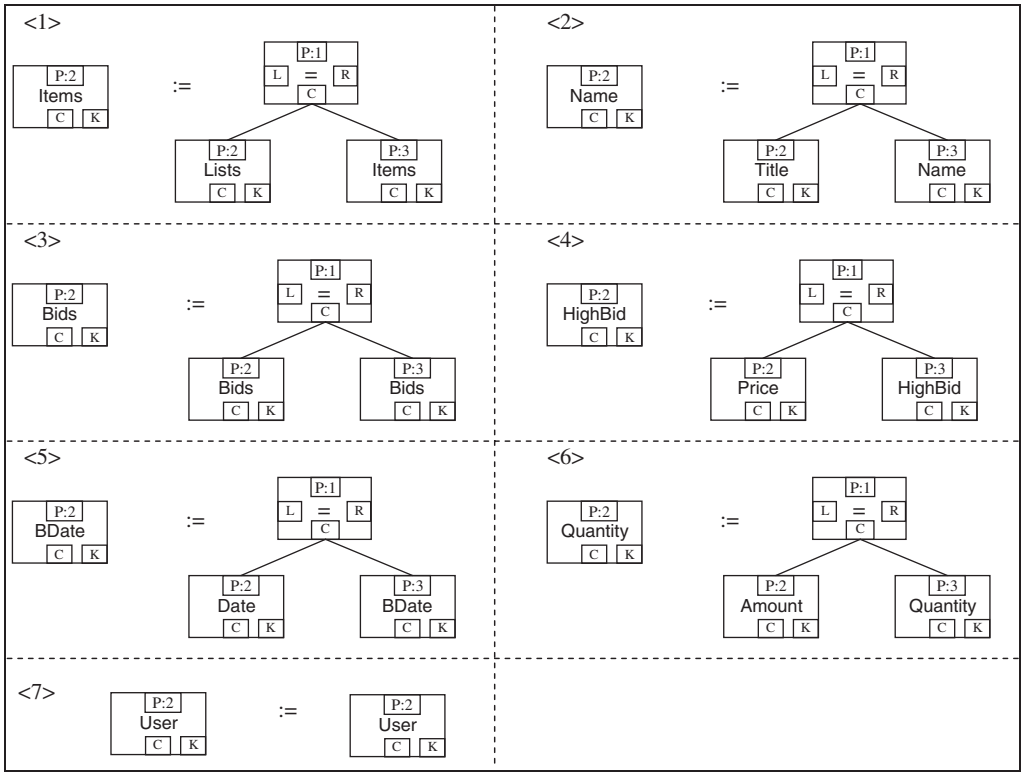

Fig. 14. Generated specific rules for the ModelGen operator.

The ModelGen operator can be described by a set of generic rules in Fig. 13. We take the mapping graph shown in Fig. 4 as our example input. The input host graph of the ModelGen operator is the result of combining a data model graph with a mapping graph as discussed in Section 4.3. The example mapping graph without links in the right model graph is the input to AutoGen. The input host graph includes a data model and a mapping between the data model and another data model, which will be generated from the ModelGen operator. The generic rules for a ModelGen operator illustrated in Fig. 14 are considered the other input to AutoGen. After parsing the input host graph according to the generic rules, AutoGen produces a specific ModelGen operator as shown in Fig. 14. The specific operator in Fig. 14 consists of 7 rules that are specific to the input data model shown in Fig. 4. Each rule outputs an element for result data model. For example, rule $\langle 1 \rangle$ produces element Items from the mapping between Items and Lists. Rules $\langle 2 \rangle$–$\langle 6 \rangle$ similarly output elements for output data model based on mappings. Rule $\langle 7 \rangle$ generates element User for output data model from input element User, because User is a new element from output data model without mapping.

For specific input host graphs, the default generated specific rules will produce the same results as results of generic rules. By tuning the specific rules, users can customize the results to meet domain specific requirements.

6. Related work

Many general purpose graph grammars and visual systems have been proposed, some of which are used to represent and manage data models [21–23]. One of the best known graph grammars is Schürr's Layered Graph Grammar (LGG) [24,25], which can be used to specify an ER data model. Schürr also proposed the Triple Graph Grammar [26] to represent and support the specification of interdependencies between graph-like data structures. Different from our approach, the TGG specifies the translation from input graphs to output graphs in a generic fashion and does not consider mappings between input graphs. Our approach explicitly defines the relationships between input graphs that represent data models, and construct graph transformation rules for each operator based on the mappings such that the operators are customizable.

Many graph grammars, including LGG, match graphs by types or a parameterized mechanism, similar to the homomorphism in this paper. The rules in these grammars are considered generic, since they can be applied to multiple inputs. It is however hard to customize these rules for specific applications. Apart from generic rules, we also provide customizable specific rules [9], which can be automatically generated by AutoGen. Jahnke and Zundorf presented *varlet*, a database reverse engineering environment based on triple graph grammars [27]. The varlet environment supports the analysis of legacy database systems, and translation of any relational schema into a conceptual object-oriented schema. It concentrates on the translation of data instances rather than on data models.

More recent work of Wermelinger and Fiadeiro [28] 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. [29]. Bézivin et al. [30] proposes a declarative translation language for model management of software architecture. These approaches are applied to software architectures. This paper however concentrates on the manipulation of data models, such as SQL schemas, XML Schemas, rather than software architectures.

Much progress in model management research has been made. For example, match has been implemented in Cupid [20,31], Clio [32], and so on [17]. Merge is also implemented by Pottinger and Bernstein [18] based on the BDK algorithm [33], and data integration project Clio [32] based on a query language specific to databases or XML schemas. Rondo [7], as the first prototype of generic model management system, implements a complete set of operators defined by Bernstein et al. [5] and extends with some new atomic operators. Rondo provides textual interfaces for users to program rather than a visual and interactive presentation supported by graph grammars. This paper does not define a set of generic rules for complex situations, such as semantics conflicts and complex mappings. The rules in Fig. 5 are only illustrated as a prototype for demonstrating the principle. One of our future works is to write a set of more powerful generic rules to be able to solve these hard problems.

7. Conclusion

This paper has formalized inputs and outputs of visual model management operators by defining the basic concepts of visual model management operators. Based on the formal definitions, this paper has proposed an automatic mechanism, called AutoGen, for generating specific rules from generic rules upon input data models and mappings.

AutoGen employs a new algorithm to generate output specific rules. The automatic process eliminates the need for manually designing specific rules and yet provides users with customizable operators for tuning the results.

Given the automatic process, the graph transformation-based model management provides two seamless levels of operators. Such a model management architecture provides generic operators and customizable specific operators for users to manipulate data models. It further allows users to construct data model-related applications with minimum effort. The approach greatly eases the usage of model management systems and enhances their applicability.

Acknowledgments

The authors would like to thank the Guest Editors and the anonymous reviewers for their insightful and constructive comments that have helped us to significantly improve the presentation.

References

[1] P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J.M., L. Serafini, I. Zaihrayeu, Data management for peer-to-peer computing: a vision, in: Proceedings of the Fifth International Workshop on the Web and Databases, Madison, WI, June 2002, pp. 89–94.
[2] H.S. Thompson, D. Beech, M. Maloney, N. Mendel-Sohn (Eds.), XML Schema Part 1: Structures. W3C Document, April 2000.
[3] M. Makoto, RELAX (REgular LAnguage Description for XML). Internet Document, April 2000, 〈http://www.xml.gr.jp/relax/〉.
[4] A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, K. Schwarzhof, Schema for object-oriented XML 2.0, W3C Document, July 1999.
[5] P.A. Bernstein, A. Halevy, R.A. Pottinger, A vision for management of complex models, SIGMOD Record 29 (4) (2000) 55–63.
[6] P.A. Bernstein, Applying model management to classical meta data problems, in: Proceedings of the 2003 CIDR Conference, Asilomar, CA, January 2003, pp. 209–220.
[7] S. Melnik, E. Rahm, P.A. Bernstein, Rondo: a programming platform for generic model management, in: Proceedings of the SIGMOD 2003 Conference, San Dieago, CA, June 2003, pp. 193–204.
[8] S. Melnik, P.A. Bernstein, A. Halevy, E. Rahm, Supporting executable mappings in model management, in: Proceedings of the SIGMOD 2005 Conference, Baltimore, MD, June 2005, pp. 167–178.
[9] G.L. Song, K. Zhang, J. Kong, Model management through graph transformation, in: Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC'04), Rome, Italy, September 2004, pp. 26–29.
[10] D.Q. Zhang, Generation of visual programming languages, Ph.D. Thesis, Macquarie University, 1998.
[11] K. Zhang, D.-Q. Zhang, J. Cao, Design, construction, and application of a generic visual language generation environment, IEEE Transactions on Software Engineering 27 (4) (2001) 289–307.
[12] D.Q. Zhang, K. Zhang, Reserved graph grammar: a specification tool for diagrammatic VPLs, in: Proceedings of the 13th IEEE Symposium on Visual Languages, Capri, Italy, 23–26 September 1997, pp. 284–291.
[13] R. Allen, D. Garlan, Formalizing architectural connection, in: Proceedings of the 16th International Conference on Software Engineering, 1994, pp. 71–80.
[14] J. Kong, K. Zhang, X.Q. Zeng, Spatial graph grammars for graphical user interfaces, ACM Transactions on Computer-Human Interaction (2005).
[15] D.Q. Zhang, K. Zhang, J. Cao, A context-sensitive graph grammar formalism for the specification of visual languages, The Computer Journal 44 (3) (2001) 187–200.
[16] K. Zhang, D.Q. Zhang, Y. Deng, Graphical transformation of multimedia XML documents, Annals of Software Engineering 12 (1) (2001) 119–137.

[17] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, The VLDB Journal 10 (2001) 334–350.
[18] R.A. Pottinger, P.A. Bernstein, Merging models based on given correspondences, in: Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003, pp. 826–873.
[19] T. Milo, S. Zohar, Using schema matching to simplify heterogeneous data translation, in: Proceedings of Very Large Databases (VLDB), New York, August 1998.
[20] J. Madhavan, A.Y. Halevy, Composing mappings among data sources, in: Proceedings of the 29th VLDB Conference, Berlin, German, September 2003, pp. 572–583.
[21] P. Bottoni, S.-K. Chang, M.F. Costabile, S. Levialdi, P. Mussio, Defining visual languages for interactive computing, IEEE Transactions on Systems, Man and Cybernetics, Part A 32 (6) (1997) 773–783.
[22] P. Bottoni, S.-K. Chang, M.F. Costabile, S. Levialdi, P. Mussio, Modeling visual interactive systems through dynamic visual languages, IEEE Transactions on Systems, Man and Cybernetics, Part A 32 (6) (2002) 654–669.
[23] G. Rozenberg, E. Welzl, Boundary NLC graph grammars—basic definitions, normal forms, and complexity, Information and Control 69 (1986) 136–167.
[24] G. Engels, C. Lewerentz, M. Nagl, W. Schafer, A. Schürr, Building integrated software development environments Part 1: tool specification, ACM Transactions on Software Engineering and Methodology 1 (2) (1992) 135–167.
[25] J. Rekers, A. Schürr, Defining and parsing visual languages with layered graph grammars, Journal of Visual Languages and Computing 8 (1) (1997) 27–55.
[26] A. Schürr. Specification of graph translators with triple graph grammars, in: International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, Spinger, Berlin, June 1994.
[27] J. H. Jahnke, A. Zudorf, Using graph grammars for building the varlet database reverse engineering environment, Technical Report tr-ri-98-201, University of Paderborn, 1998.
[28] M. Wermelinger, J.L. Fiadeiro, A graph transformation approach to software architecture reconfiguration, Science of Computer Programming 44 (2002) 133–155.
[29] G. Engels, R. Heckel, J.M. Küster, L. Groenewegen, Consistency-preserving model evolution through transformations, in: UML'02, Lecture Notes in Computer Science, vol. 2460, Springer, Berlin, 2002, pp. 212–227.
[30] J. Bézivin, E. Breton, G. Dupé, P. Valduriez, The ATL transformation-based model management framework, Research Report No. 03.08, Université de Nantes, September 2003.
[31] J. Madhavan, P.A. Bernstein, E. Rahm, Generic schema matching using cupid, in: Proceedings of the 27th VLDB Conference, Rome, Italy, September 2001, pp. 49–58.
[32] R.J. Miller, M.A. Hernandez, L.M. Haas, L. Yan, C. Ho, R. Fagin, L. Popa, The Clio project: managing heterogeneity, SIGMOD Record 30 (1) (2001) 78–83.
[33] P. Buneman, S.B. Davidson, A. Kosky, Theoretical aspects of schema merging, in: Proceedings of the Third International Conference on Extending Database Technology, Vienna, Austria, March 1992, pp. 152–167.