ELSEVIER

Science of Computer Programming 45 (2002) 99–136

Science of Computer Programming

www.elsevier.com/locate/scico

Supporting iterations in exploratory database reengineering processes

Jens H. Jahnke[a,*], Wilhelm Schäfer[b], Jörg P. Wadsack[b], Albert Zündorf[c]

[a] *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC V8W 3P6, Canada*

[b] *Software Engineering Group, Department of Mathematics and Computer Science, University of Paderborn, D-33095 Paderborn, Germany*

[c] *Department of Mathematics and Computer Science, Technical University of Braunschweig, P.O. Box 3329, D-38023 Braunschweig, Germany*

Abstract

Key technologies like the World Wide Web, object-orientation, and distributed computing enable new applications, e.g., in the area of electronic commerce, management of information systems, and decision support systems. Today, many companies face the problem that they have to reengineer pre-existing information systems to take advantage of these technologies. Various computer-aided reengineering tools have been developed to reduce the complexity of the reengineering task. A major limitation of current approaches, however, is that they impose a strictly phase-oriented, waterfall-type reengineering process, with little support for iterations. Still, such iterations often occur in real-world examples, e.g., when additional knowledge about the legacy system becomes available or when the legacy system is modified during an ongoing migration process. In this paper, we present an approach to incremental consistency management that allows to overcome this limitation in the domain of database systems by integrating reverse and forward engineering activities in an intertwined process. The described mechanism is based on a formalization of conceptual schema translation and redesign transformations by graph rewriting rules and has been implemented and evaluated with the Varlet database reengineering environment.

Keywords: Database reengineering; Design recovery; Schema redesign; Computer-aided software engineering; Process iterations; Legacy information system; Database migration; Graph grammars

* Corresponding author.

E-mail addresses: jens@acm.org (J.H. Jahnke), wilhelm@uni-paderborn.de (W. Schäfer), maroc@uni-paderborn.de (J.P. Wadsack), zuendorf@ips.cs.tu-bs.de (A. Zündorf).

0167-6423/02/$ - see front matter
PII: S0167-6423(02)00056-4

1. Introduction

Effective and efficient information management is a crucial factor for the competitiveness of today's companies. It enables them to respond quickly to changing conditions on a global market. Key technologies like the *World Wide Web* (Web), *Object-Orientation* (OO), *Client/Server applications*, and *open system standards* (e.g., CORBA [52]) greatly influence modern business processes. In addition to new applications in the area of electronic commerce, there has been increasing interest in using enterprise-wide data access to build management information systems and decision support systems. While new company start-ups are able to purchase information systems (IS) that take advantage of the latest technology, more established enterprises have to modify pre-existing IS to fit new requirements and exploit the new technologies. This is often a challenging task because many IS have evolved over several generations of programmers and have obsolete documentation or none at all. Such applications are usually called *legacy IS* [51].

In this decade there has been an increasing effort to develop concepts and methods to reengineer legacy IS. Some of these methods have been implemented in computer-aided reengineering (CARE) tools to automate laborious activities and reduce the complexity of the reengineering problem. As the persistent data structure is the central part of a legacy IS [1], many approaches focus on *database schema analysis* [22,45,38,47] (reverse engineering) and/or *schema translation and redesign* (forward engineering) [5,16,39,24,14,34].

One of the most important limitations of current database reengineering (DBRE) tools is that they do not consider the evolutionary and exploratory nature of the reengineering process [19]. They impose a strictly phase-oriented, waterfall-like reengineering process, without the support for *iteration*. This is an important limitation in practice, as iterations between schema analysis and redesign steps occur frequently: when a reengineer learns more about the abstract design of a legacy database, (s)he often refutes some initial assumptions and does further investigations. Moreover, migration projects might have durations from several months up to years. It is probable that urgent requirements demand (on-the-fly) modifications of the original database during this period. These modifications have to be reflected in the migrated target system, which, of course, leads to the demand for iteration in the reengineering process. Unfortunately, iterations are hardly supported by current tools, i.e., most reengineering tools do not allow the propagation of incremental changes of the legacy system (respectively updates of the known information *about* a legacy system) to a representation of the system that has already been partly reengineered. In practice, this limitation often forces tool users to start over again with automatically created design models of the changed legacy system. However, in doing this, they lose all work they have performed in interactively redesigning and refactoring the previous representation of the legacy system. In this paper, we describe an integrated CARE environment that overcomes the described problem. Our approach is based on a formalization of conceptual translation and redesign operations by *graph productions* [40]. This formalization allows us to achieve an incremental consistency management, i.e., to integrate schema analysis and redesign activities in an iterative and interactive reengineering process.

The rest of this paper is structured as follows. In the next section, we introduce our approach with a case study that motivates the need for tools supporting an iterative reengineering process. Section 3 discusses related work and Section 4 describes the basic data structures employed in our migration environment. Based on these data structures, Section 5 introduces schema mapping rules that allow an incremental translation from *logical* to *conceptual schemas* and vice versa. Section 6 proposes redesign transformations that allow adaption and extension of the resulting conceptual schema in order to support new requirements. In contrast to most other approaches our redesign transformations do not just edit the conceptual schema but also they maintain the correspondence to the original logical schema which allows for the development of an incremental change propagation mechanisms to support process iterations (Section 7). Finally, Section 8 closes with some concluding remarks about our experiences with a medium-sized industrial project.

2. Motivating example application

The following case study reflects some of our experiences with an industrial project. It deals with a legacy *product and document information system* (PDIS) of an international enterprise that produces a great variety of drugs and other chemical products. Traditionally, this system has been used by members of the central hotline at the company headquarter. PDIS is based on a relational database management system (Oracle). The part of PDIS that deals with data management functionality comprises lines of mixed C and C++ code. Most queries had been implemented by passing dynamic SQL text strings to Oracle DBMS library functions. Only a small number of queries had been implemented in embedded SQL. The database contained information on approximately 100,000 products in 85 tables and 347 attributes. Now, the IT department plans to employ network-centric technology to establish an integrated Web-based *marketing information system* based on the existing PDIS. The aim of this project is to reduce costs and increase the availability of current product data (24 hours a day). In addition, the new Web-based system should be implemented in an object-oriented, platform independent programming language (Java and C++). This would facilitate its extensibility concerning future requirements, e.g., enterprise wide information integration.

In order to implement the desired marketing information system the legacy relational database schema has to be well understood. Unfortunately, the legacy database has little documentation and the programmers have left the company. Thus, PDIS has to be *analyzed* to yield a semantically enriched logical schema, which can be *translated* (and *redesigned*) into a conceptual (object-oriented) target schema that is suitable for the new marketing information system. In general, this is an iterative and explorative process, as indicators for abstract design concepts are often hidden in different parts of the legacy database, including its data, code, physical schema, and its (obsolete) documentation (cf. Fig. 1). Furthermore, many legacy databases comprise arcane coding concepts (e.g., variant records) and various kinds of optimization structures.

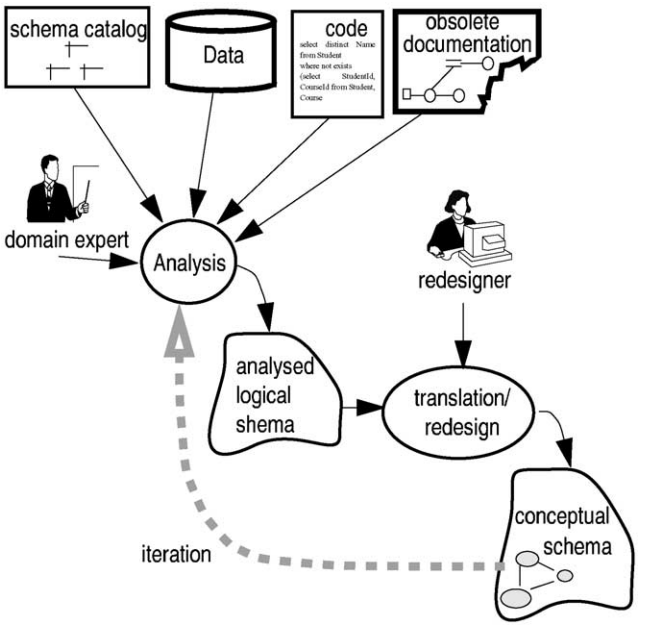

Fig. 1. Schema reengineering process.

The following description demonstrates how this evolutionary process is supported by our CARE environment *Varlet*, i.e., how *Varlet* preserves the consistency between the analyzed source schema and the redesigned conceptual target model.

2.1. Iteration 1: schema analysis, translation, and redesign

Fig. 2 shows a screenshot of the *Varlet analyzer view* that displays a detail of the legacy database schema that includes 85 tables and 347 attributes in total. Each box represents a relational table. Foreign keys are represented as directed lines with the inclusion sign "⊂" in the triangles showing the relation between these boxes. If a foreign key consists of more than one attribute, the correspondences of attributes in different tables are marked by numbers. Attributes which belong to primary keys are displayed in bold face.

Foreign key constraints and alternative keys are rarely specified explicitly in schema catalogs of old databases. In case of our case study, let us assume the reengineer has already recovered the constraints shown in Fig. 2 (see [27,29] for more information about the actual analysis process in *Varlet*). Moreover, the reengineer has added further semantic information about the legacy schema. For example, the equal sign "=" in the triangles between tables *ProductGroup* and *CommodityGroup* denotes that the corresponding foreign key implies an inclusion dependency (IND) [14] in *both* directions. This means that for each tuple in table *CommodityGroup* there has to be at least one tuple in table *ProductGroup* with equal value in column *CG*.

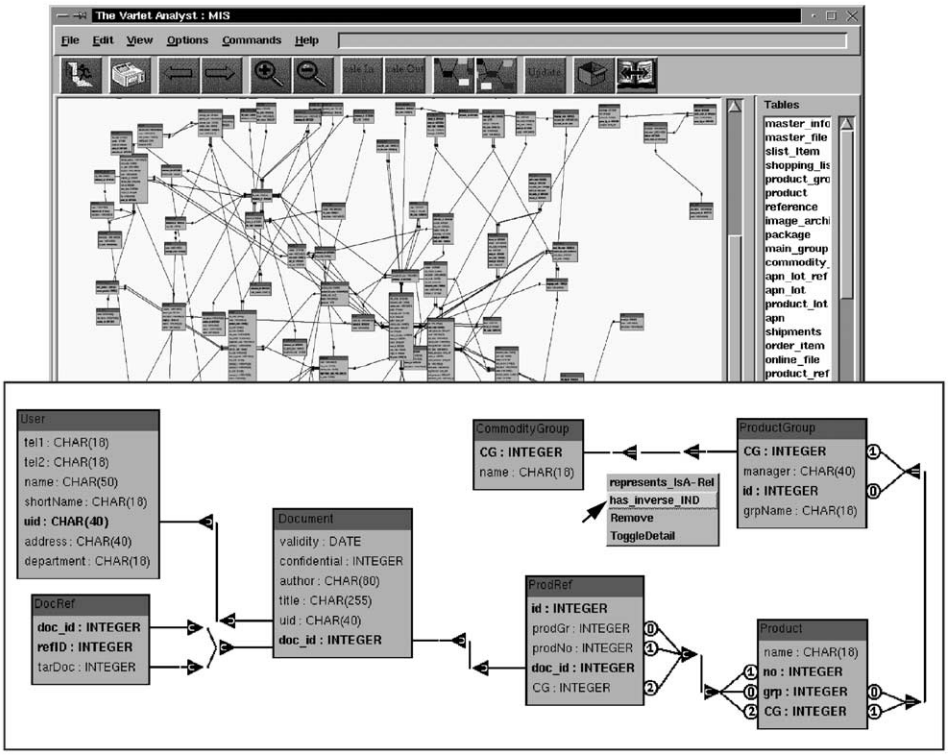

Fig. 2. Case study: analyzed relational legacy schema (detail).

After the PDIS has been analyzed, *Varlet* automatically translates the semantically enriched schema into an object-oriented conceptual model. This initial conceptual schema can be redesigned and extended by using a set of pre-defined *redesign operations*. Fig. 3 shows a sample resulting conceptual model for our scenario. Bold gray arrows denote ten sample applications of redesign operations the reengineer has performed after the initial translation. For example, the initially created class *User* has been renamed (1) to *StaffMember* and has been generalized (2) by a new (abstract) class *User* with a specialization (3) *Customer* that has a new attribute (4) *company*. Subsequently, the reengineer aggregates attributes *tel1* and *tel2* into a complex attribute (5) and splits class *StaffMember* (6). Then, attribute *doc_id* is removed from class *Document* (7), as artificial keys are not needed in the object model. Finally, the reengineer transforms class *DocRef* into an ordered association (8) and changes the cardinality of its left-hand side (9). The dialog in the bottom right corner of Fig. 3 exemplifies that redesign operations are interactively invoked by actualizing their formal parameter lists. In analogy to the redesign operation (8), the reengineer transforms class *ProdRef* into an ordered association (10). The functionality portrayed above is supported by several existing database reengineering and evolution tools, e.g., DB-Main [13]. However, the problems start whenever iterations occur in the reengineering

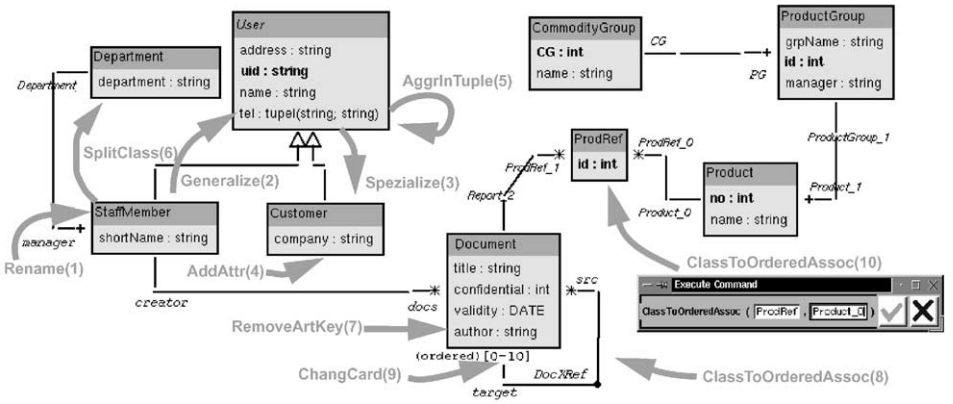

Fig. 3. Redesigned conceptual model.

variant	id	prodNo	prodGr	doc_id	CG	*semantic interpretation*
1	...	...	...	...	...	*cross reference between documents and products*
2	...	null	...	...	...	*cross reference between documents and product groups*
3	...	null	null	...	...	*cross reference between documents and commodity groups*
4	...	null	null	...	null	*place holder for document cross references (stored in table DocRef)*

Fig. 4. Variants of table *ProdRef*.

process. Unfortunately, such iterations happen frequently in practice. This problem is illustrated in the next paragraph.

2.2. Iteration 2: schema completion and retranslation

Let us assume that from a later investigation of the legacy data in table *ProdRef*, the reengineer notices that many rows comprise attributes with null-values. (S)he discovers that there are actually four recurring "variants" of rows in this table (cf. Fig. 4). By talking to PDIS users, the reengineer learns that table *ProdRef* not only maintains cross references from documents to products but also to product groups and commodity groups. Moreover, (s)he learns that all cross references, either between different documents (table *DocRef*) or between a document and products (table *ProdRef*), have unique numbers with respect to the referencing document. Consequently, (s)he discovers that for every row in table *DocRef* there exists a row (of Variant4) in table *ProdRef* with equal key values. In addition, the reengineer finds out that *shortName* is an alternative key of table *User*, borrowed by table *ProductGroup* in column *manager*.

The updated and completed relational schema is given in Fig. 5. In this figure, we used ovals to mark the differences between the completed relational schema and the

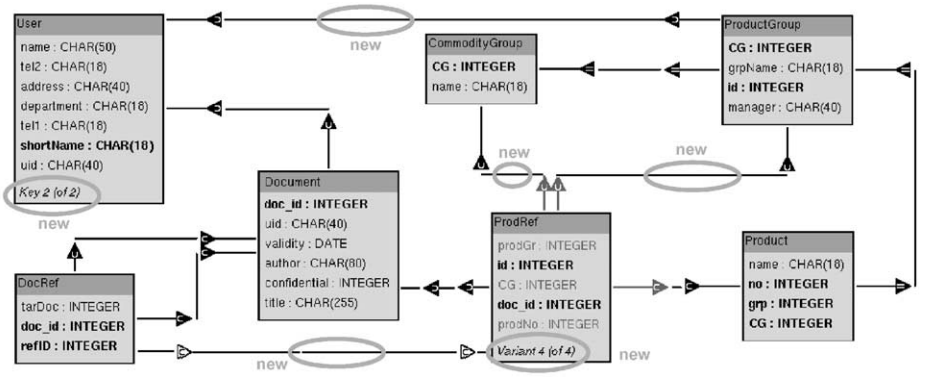

Fig. 5. Completed relational schema.

first analysis result (cf. Fig. 2). Now, the graphical representation of table *ProdRef* contains an annotation that there are four different "variants" of entries in this table. The reengineer can browse through these variants, while null-columns (and disabled foreign keys) are dimmed to gray. The white triangles at the connection between *DocRef* and *ProdRef* represent the semantic information that the corresponding foreign key has been classified as an inheritance relationship. (To simplify the layout of Fig. 5, we selected a more abstract representation of foreign keys that hides actual attribute correspondences.)

Now that the information about the legacy source schema has changed, consistency with the previously created conceptual model has been lost. Using existing waterfall-oriented tools, the reengineer has two options to reestablish consistency: (1) (s)he starts the redesign process all over again with a newly generated initial translation of the modified legacy schema, or (2) (s)he tries to manually determine the impact of the modifications on the redesigned conceptual model. In practice, both alternatives are unsatisfactory for larger schemas: the first solution will most likely force the reengineer to manually redo many redesign operations, which (s)he has already performed once, while the second solution is prone to error.

Our approach overcomes this problem by providing an incremental mechanism to reestablish consistency between the legacy schema and the redesigned conceptual model. Fig. 6 shows the new conceptual model, which has been updated automatically according to the new information about the legacy schema. The environment determined automatically that operations 8, 9 and 10 have to be undone (cf. Fig. 3). Operations 8 and 10 are no longer applicable, because the different variants of table *ProdRef* have been mapped to an inheritance structure by the initial translation and this violates the pre-condition [1] of operation *ClassToOrderedAssoc*. Operation 9 is no longer applicable, as it depends on the applicability of operation 8.

[1] A class can only be transformed to an association if this class has exactly two associations, since the newly analyzed IND classes *DocRef* and *ProdRef* have more than two associations.

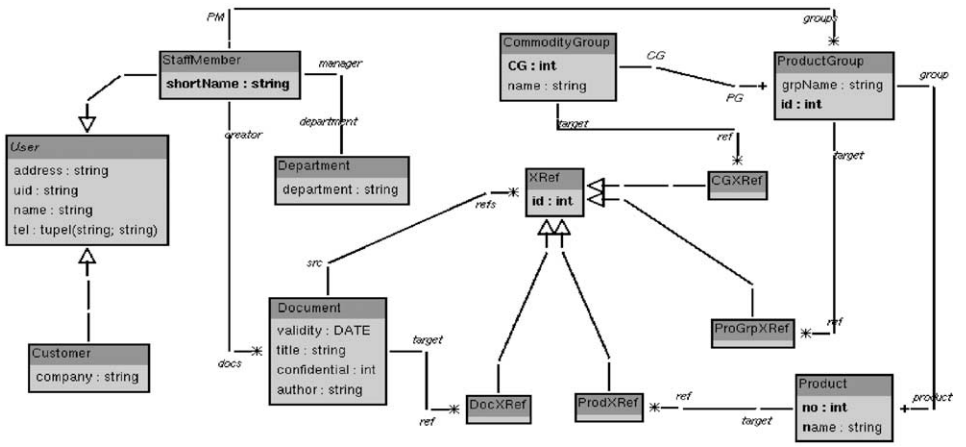

Fig. 6. Updated conceptual schema.

This case study shows only one iteration between analysis and redesign activities. However, in general, database reengineering is an evolutionary process and deals with many iterations. A consistency management mechanism is thus a key component of a DBRE environment. In the following sections, we will describe implementation concepts for such a component. The main idea is to specify schema translation and redesign operations as graph productions with formally defined pre- and post-conditions. If the legacy schema has been modified, input/output-dependencies between such rules are used to determine those operations that are affected by the modification. Subsequently, all affected operations with violated pre- or post-conditions are undone.

The sketched approach to incremental schema reengineering is part of a larger reengineering process including additional activities for schema analysis and middleware generation (Fig. 7). The analysis step and its support by *Varlet* is described in [27,29,28]. The data integration step and its support by *Varlet* is explained in [25]. The schema migration process described in this paper starts with a canonical and automatic translation of the analyzed logical schema into a conceptual data model. Then, the resulting conceptual schema is redesigned and extended interactively by the reengineer.

3. Related work

The literature contains various algorithms for canonical translation of logical to conceptual database schemas (e.g., [36,9,31,35,48,2,37,34,39,16]). However, reengineering practitioners have criticized that these fully automatic approaches often make unrealistic assumptions about the quality of the legacy system. They provide too little flexibility to be usable in many practical reengineering scenarios. Premerlani and Blaha emphasize that a flexible, interactive approach to DBRE is more likely to succeed than batch-oriented compilers [38]. Vossen and Fahrner suggest to combine an initial automatic translation with a subsequent manual *redesign phase* [14]. However, they do not

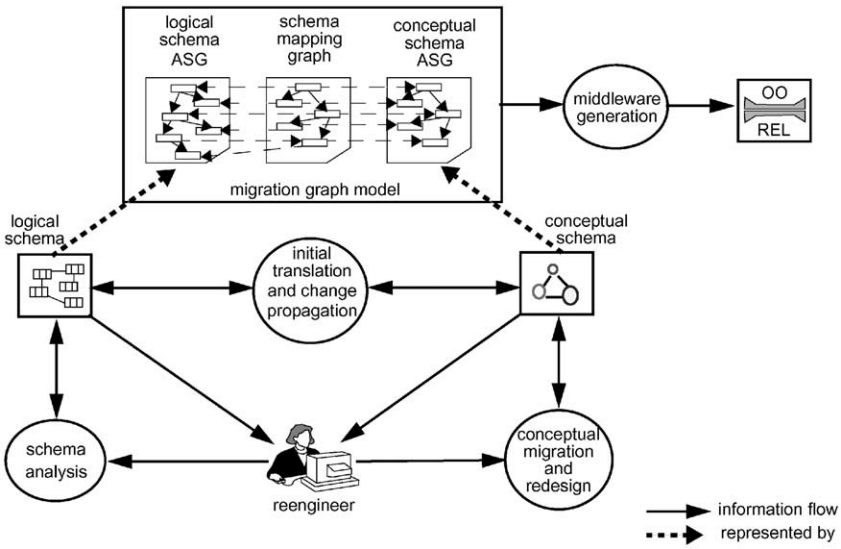

Fig. 7. Incremental schema migration and generative data integration.

provide any tool support for this human-intensive activity. In [5], Behm et al. propose an interactive schema migration environment that provides a set of alternative schema mapping rules. In this approach, which is similar to our migration environment in its early stages [26], the reengineer repeatedly chooses an adequate mapping rule for each schema artifact that has to be mapped. Hainaut et al. skip the initial translation step completely and use a common generic data model that subsumes conceptual constructs as well as logical (and physical) constructs [20,17]. Based on this common data model, Hainaut et al. have defined a catalog of schema transformations which are used to gradually replace low-level implementation constructs by more abstract concepts [21]. However, the execution of *in-place* transformations (suggested by Hainaut) impede iterative DBRE processes because the original (logical) schema implementation is lost during the migration process. Jeusfeld and Johnen propose an approach to schema migration that employs a generic *meta model* as mediator [32]. This meta model includes general modeling concepts like objects, types, and links with different cardinality. The schema migration process is performed as follows. In a first step, the concepts of the concrete data model of the legacy database is classified in terms of concepts of the meta model. The same is done for the target data model. The classification of the source data model is the basis to map all schema artifacts to equivalent artifacts in the meta model. Analogously, the classification of the target data model is used to map this meta schema back to an equivalent schema in the target data model. These mapping steps are performed in an interactive process and the tool prompts the reengineer when ambiguities arise.

Premerlani and Blaha propose a set of simple, loosely coupled tools for textual search and data analysis, e.g., grep, awk-scripts, and pre-defined database queries [38]. Their

DBRE process is based on the *Object Modeling Technique (OMT)* [41] and starts with an initial object model where each relational table represents a candidate class. Subsequently, the reengineer has to detect abstract design concepts based on a set of informal heuristics, guidelines, and clues [8]. Several case studies have shown that this approach provides the necessary flexibility required to reengineer real-world legacy systems. However, since this approach is based on loosely coupled tools, it provides very little support for iteration in the reengineering process. In particular, the tool set lacks the ability to detect and propagate inconsistencies between a (modified) legacy system and its conceptual representation. Our approach to a user-centered, interactive reengineering was largely motivated by the observations described by Blaha and Premerlani. Moreover, we have implemented many of the conceptual redesign transformations proposed by these authors [8]. However, in order to support process iterations and change propagations, we employ an extensible and integrated environment, as opposed to a set of loosely coupled tools.

4. The migration graph model

Internally, the logical schema and the conceptual schema are represented by their *abstract syntax graphs* (ASGs). The dependencies between both schemas are represented by an intermediate graph called the *schema mapping graph* (SMG) (cf. Fig. 7). In case of process iterations, the information maintained in the SMG is employed to control incremental change propagation operations that reestablish project consistency. The entire graph (both ASGs and the SMG) is called *migration graph*.

The formal basis for the migration graph is the concept of a *directed, attributed graph with node and edge types* [12]. We use the term *graph* for abbreviation whenever we refer to a directed, attributed graph with node and edge types. Such a graph is defined as follows.

Definition 1. *Graph*: $G := (N, E, t_N, A)$ is a *graph* over two given type label sets L_N, L_E with:

- $N(G) := N$ is a finite set of *nodes*;
- $E(G) := E \subseteq N \times L_E \times N$ is a finite set of edges;
- $t_N(G) : N \to L_N$ is a *typing function* for nodes;
- A is a finite set of *node attributes*, each $a \in A$ is a partial function $a : N \to dom(a)$, where "dom(a)" denotes the domain of attribute "a".

Moreover, we define the following auxiliary functions:

- $s : E \to N$ and $t : E \to N$ with $s((n_1, l, n_2)) := n_1$ and $t((n_1, l, n_2)) := n_2$, return for each edge $(n_1, l, n_2) \in E$ its *source* and *target*;
- $t_E : E \to L_E$ returns for each edge $(n_1, l, n_2) \in E$ its label l.

In the following, we are not interested in defining particular instances of migration graphs but we aim on defining a schema for a graph class that contains all valid migration graphs. We call such a schema a *graph model*. We have used the formal specification language *PROgrammed Graph REplacement Systems*

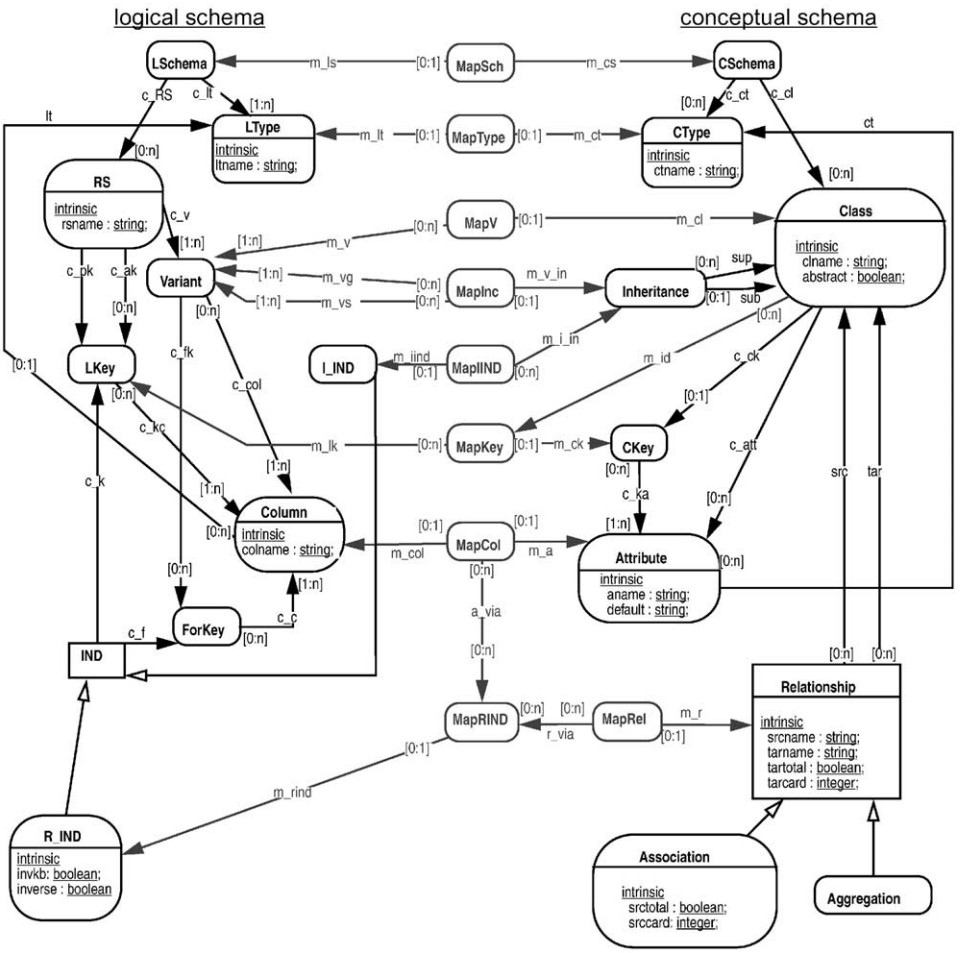

Fig. 8. Migration graph model.

(*Progres*) [44] to define and implement the graph models discussed in this paper.

Fig. 8 shows the most important parts of this graph model in a diagrammatic *Progres* notation that is similar to UML [42]. To avoid confusion with classes and associations which are modeled within a conceptual database schema, we keep on using the graph-oriented terms *node type* and *edge type* instead of *class* and *association* (used in UML). Note, that cardinalities of edge types are denoted in form of intervals. If no cardinality is specified in the diagram its default value is defined as [1:1]. A formalization of the complete migration graph model in form of a textual *Progres* graph schema is given in [25].

The left-hand side of Fig. 8 represents the ASG model for the analyzed logical schema. Starting with the root node labelled *LSchema* it consists of a number of

relation schemas[2] (*RS*) and column types (*LType*). RS in turn consist of a non-empty set of *Variant* nodes, a primary key (*LKey*) that is referenced by a *c_pk* edge, and a set of alternative keys which are referenced by edges of type *c_ak*. Each *Variant* node contains a set of foreign-keys (*ForKey*) and a non-empty set of columns (*Column*). A column has an *lt* edge to point to its type. A inclusion dependency (*IND*, e.g., foreign key) between tables can be classified semantically as an inheritance (*I-IND*) (cf. Section 2.2) or a normal reference (*R-IND*). This classification of INDs is based on [14].

The right-hand side of Fig. 8 depicts the ASG model that specifies the chosen conceptual model. Starting with the root node *CSchema* the model describes the syntactic structure of a typical EER-model. Such a model contains classes, attributes, attribute groups which form keys, inheritance, and the possible relationships between classes, namely associations and aggregations.

The SMG connects the ASGs of the logical and the conceptual schema and represents their interdependencies. The graph elements of the SMG model are displayed in gray color in Fig. 8. The information maintained in the SMG serves two distinct purposes: (1) it is the basis for the initial schema translation and (2) it enables the generation of schema mapping descriptions for middleware components that facilitate data integration [25]. The SMG model is rather complex because it has to provide suitable flexibility to allow for alternative schema mappings.

A node of type *MapSch* is used to connect the syntactic roots of both ASGs. *MapType* nodes are used to map column types to attribute types. Each variant in the logical schema is represented by a concrete class in the conceptual schema. If an RS has more than one variant, these variants usually comprise common columns which imply an inheritance hierarchy with *abstract* classes in the conceptual schema. Consequently, an abstract class is mapped to more than one variant, namely all variants which are represented by its *concrete* subclasses. In the SMG, correspondences between classes and variants are represented by nodes of type *MapV* (cf. Fig. 8).

Inheritance relationships in the conceptual model can be mapped in two different ways to constructs in the logical schema. Firstly, they can be mapped to the *inclusion* of more specific variants in less specific variants that belong to the same RS. Consider table *ProdRef* in Fig. 4 as an example for such a situation. In this example, Variant 4 of table *ProdRef* is less specific than Variant 3, i.e., Variant 4 is included in Variant 3. This situation is represented by an inheritance relationship in the conceptual model which is mapped by a node of type *MapInc* to the two variants (cf. Fig. 9). An edge of type *m_vs* is used to reference the variant which is more specific, while an edge of type *m_vg* references the variant which is more general. A formal technique for building the inheritance lattices based on the subsumption of common features is discussed in [46]. Secondly, the migration graph allows us to map inheritance relationships to INDs in the logical schema that have been classified as inheritance relationships (I-INDs) in the analysis process. In the latter case, the mapping is represented by a node of type *MapIIND*.

[2] Note, that in the database literature the term *relation schema* is used to denote the structure of a single database table (as opposed to the term *relational database schema*, which denotes the schema of an entire database).

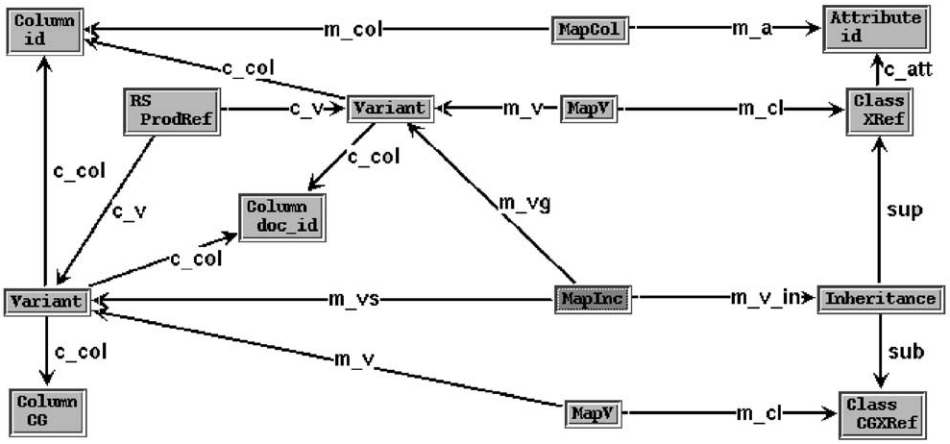

Fig. 9. Sample situation: correspondence between variant and inheritance structures.

Nodes of type *MapKey* are used to map primary keys in the logical schema to keys in the conceptual schema. According to the ODMG data model, our conceptual model includes the notion of unique object identifiers (OIDs) for instances of classes [10]. Hence, it is not required that every class contains a value-based key. Still, if we aim for object-relational data integration, OIDs have to be resolved to value-based keys in the logical data model. For this purpose, every class has an edge of type *m_id* that references a *MapKey* node in the SMG.

Attributes are mapped to columns by nodes of type *MapCol*. To provide the flexibility required for different alternative schema mappings, we admit that attributes of a single class can be mapped to columns in separate RS. For such "remote" columns, the SMG has to maintain the access path from the RS that includes the value-based key associated to the class and the RS which includes the remote column. This information is represented by edges of type *a_via*: if a *MapCol* node does not have an *a_via* edge the mapped column belongs to the RS that contains the key referenced by the *m_id* edge of the class that contains the mapped attribute. Otherwise, the mapped column belongs to a different RS and the *a_via* edge of the corresponding *MapCol* node points to a set of *MapRIND* nodes. These nodes represent the access path from the RS that contains the key referenced by the *m_id* edge to the RS that contains the mapped column. Each *MapRIND* node is connected to a *R-IND* node which logically represents a foreign key that has to be traversed in order to access the mapped column. In analogy to the mapping of columns, *MapRel* nodes and *r_via* edges are used to map associations and aggregations to sets of foreign keys (represented by nodes of type *R-IND*).

5. Schema translation using triple graph grammars

Most existing approaches to conceptual schema translation employ *rule-based* transformation systems. Such transformation rules are often specified in a textual pattern

language [34] or in a calculus which is based on first-order logic and set theory [5,20]. However, despite their precise semantics such transformation rules are difficult to understand. Therefore, researchers typically employ diagrams to explain the meaning of transformation rules. Furthermore, some formal specifications cannot be executed directly but have to be implemented in a programming language on a lower level of abstraction. In our approach, we employ *graph grammars* [40] to specify schema transformations because they are executable and have the expressiveness of diagrams.

5.1. Graph Grammars

In analogy to classical (textual) grammars, a graph grammar consist of a *start graph* and a set of (*graph*) *productions*. In general, a graph production can be defined as a pair of graphs, a set of *application conditions*, and a set of attribute *transfer clauses* (cf. Definition 2)[3]. The two graphs are called the *left-hand side* and the *right-hand side* of the production, respectively. A production r is applicable to a given *host graph* if the host graph contains a match for the left-hand side of r. The application of a production r to a host graph G replaces the match for the left-hand side of r by its right-hand side. We denote $G^{\downarrow(r,m)}$ for the graph that is produced by the application of a production r to a graph G (in a match m). The application semantics of a production to a given host graph is described in the Appendix A (Definition 4).

Definition 2. *Graph production*: A *graph production* is a tuple $r : (P, Q, C, T)$, where

- $P(r) = P$ and $Q(r) = Q$ are two graphs over the same sets of node and edge type labels; $P(r)$ is called the *left-hand side* and $Q(r)$ is called the *right-hand side* of r.
- C is a set of *application conditions*.
- T is a set of attribute *transfer clauses*.

Fig. 10 shows a simple *Progres* production *AddRSToLSchema* which specifies the extension of a logical schema by a new RS. The left-hand side of production *AddRSToLSchema* contains only a single node of type *LSchema*. If the production is applied this node is preserved because it occurs on the right-hand side with an identical node number. Furthermore, G is extended by three new nodes and three new edges which represent a new RS with one variant and a primary key.

Usually, a graph grammar is used to define a *single* graph model in terms of all possible graphs that can be derived by applying the productions to a given start graph. This approach is less suitable for specifying the mapping between two different ASG models as needed in our application. In [33], Lefering and Schürr propose an extended formalism called *triple graph grammars* that is dedicated to this problem.

[3] Note, that *Progres* productions allow for extended concepts like optional nodes, node sets, path expressions, etc. [40]. However, the semantics of these extended concepts can be defined based on the primitive concepts described here.

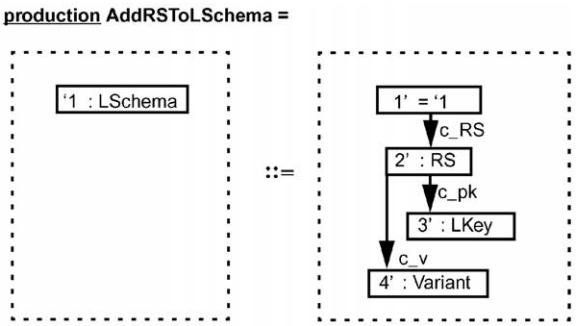

Fig. 10. Graph production *AddRSToLSchema*.

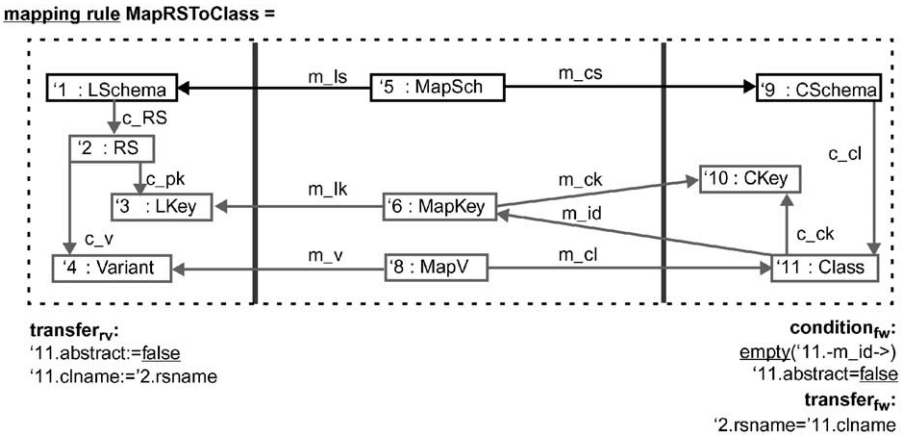

Fig. 11. Mapping rule *MapRSToClass*.

5.2. Triple graph grammars

A triple graph grammar consists of a set of *mapping rules*. Basically, each mapping rule consists of a production triple, i.e., it contains three productions. Two of these productions specify equivalent extensions of the first and the second ASG, while another production is used to extend a mapping graph that represents the correspondences between both ASGs.

Fig. 11 shows an example for such a mapping rule. The three productions are separated by vertical, gray bars. Triple graph grammars deal with *extending* productions only, i.e., no graph elements are removed. Hence, a single graphical diagram can be used to represent both sides of an extending production in a condensed way (this notation stems from [26]). For example the left production of the mapping rule in Fig. 11 is a condensed notation for production *AddRSToLSchema* in Fig. 10. The entire mapping rule *MapRSToClass* in Fig. 11 specifies that an extension of a logical

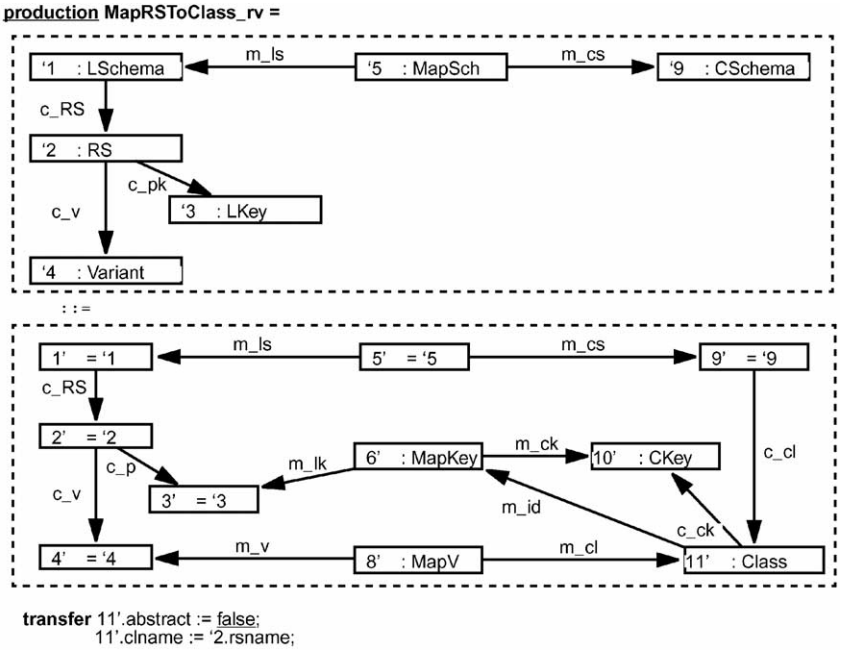

Fig. 12. Reverse production *MapRSToClass_rv*.

schema by a new RS corresponds to the extension of the conceptual schema by a new class. The production in the middle part of the mapping rule is used to update the SMG that represents the correspondence between both ASGs.

A triple graph grammar allows to generate automatic translators that create conceptual schemas from logical schemas (*reverse mapping*) and vice versa (*forward mapping*). Such an automatic translator consists of a set of conventional graph grammar productions. Each such production is derived from one mapping rule specified in the triple graph grammar. A *reverse production* p_{rv} is derived from a mapping rule by choosing its black parts and its left side as the left-hand side of p_{rv} and the elements in the entire mapping rule as the right-hand side of p_{rv} (cf. Fig. 12). In analogy, the corresponding *forward production* p_{fw} is derived by choosing the black parts and the right-hand side of the mapping rule as the left-hand side of p_{fw} and the elements in the entire mapping rule as the right-hand side of p_{fw} (cf. Fig. 13).

As defined in Definition 2, *Progres* productions might include *attribute transfer clauses*. They are added in textual form under the graphical part of the production. The first attribute transfer clause in Fig. 12 assigns the boolean value *false* to attribute *abstract* of the new *Class* (node $'11$). The second transfer clause transfers the name of the mapped RS to this new node. In a triple graph grammar, we add transfer clauses (and application conditions) for both derivable productions to each mapping rule. This is depicted in Fig. 11 where we use the suffixes rv and fw to denote whether the clauses belong to the reverse or the forward production.

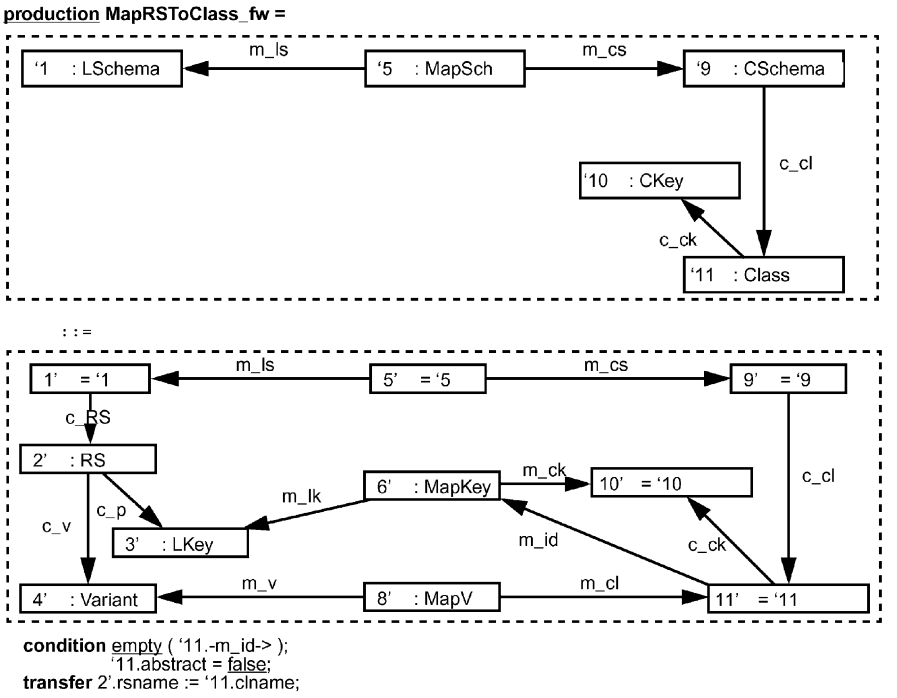

Fig. 13. Forward production *MapRSToClass_fw*.

In *Progres*, *application conditions* often contain the so-called *path expressions* [49, p. 33]. Path expressions consist of a sequence of edge traversals separated by dots, e.g., $-e1- > \cdot < -e2-$ defines a path over an outgoing edge of type *e1* and an ingoing *e2* edge. When a path expression is applied to a node n (or a set of nodes) its application returns all nodes that can be reached from n by traversing the specified path. For example, expression $'11.-m\_id- >$ in the condition part of *MapRSToClass_fw* (Fig. 13) returns all variant nodes that can be reached from node $'11$ over an outgoing edge of type *m_id*. The boolean predicate *empty* returns *true* if and only if its argument is an empty set. This condition is necessary to enable that *several* classes in an inheritance hierarchy can be mapped to variants of the *same* RS: new RS nodes are created for classes only if they do not have the same value-based key (referenced by edge *m_id*) as another class which has been mapped before. Moreover, the attribute condition "$'11.abstract = false$" ensures that only concrete classes are mapped using this production. (Otherwise different mapping rules apply.)

In this subsection, we have shown only one triple graph grammar rule that maps RS to classes. Appendix B (Examples for Mapping Rules) includes two additional examples for triple graph grammar rules which map columns and R-INDs. All eleven triple graph grammar mapping rules that have been implemented in Varlet are documented in [25].

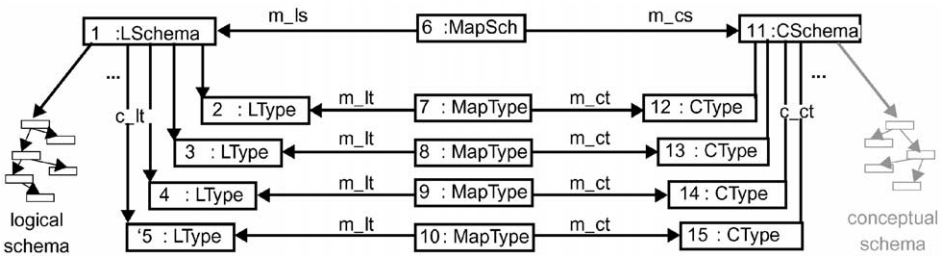

Fig. 14. Start graph for schema translation.

5.3. Schema translation

In analogy to start symbols of conventional textual grammars, graph grammars are applied to an initial graph that is called *start graph*. In our application, the minimal start graph consists of the syntactic root nodes for the ASGs of both schemas and graph elements that represent all attribute and column types (cf. Fig. 14). Pairs of equivalent atomic data types are mapped by nodes of type *MapType*. The correspondences between atomic types in the logical and the conceptual schema, respectively, depends on the concrete application context of the DBRE tool. Different DBMS provide different data types. Hence, in our approach, the reengineer has to enter atomic type correspondences in an initial customization dialog of our DBRE tool. (In some cases, it might also be necessary to implement type conversion functions. In principle, such functions can be stored in further attribute of MapType nodes. However, we abstract from this detail in the following discussion.)

In typical DBRE scenarios, the start graph contains further parts of an analyzed logical schema ASG which are going to be translated to conceptual schema constructs. Moreover, during the migration process it often occurs that modifications in conceptual schemas have to be remapped to the original logical schema. In this case, the mapping algorithm is applied to a start graph that contains ASG elements from the logical schema as well as from the conceptual schema (illustrated by the gray subgraph in Fig. 14).

5.4. The schema translation algorithm

The translation process is based on the execution of forward and reverse productions that are derived from each mapping rule. The corresponding translation algorithm is described in Fig. 15. It iteratively chooses a production r from the set of all derived productions R that has a match in the current migration graph G. Furthermore, it is validated that this match cannot be extended to a match that includes all SMG elements on the right-hand side of r. This is to avoid multiple applications of the same production in the same match. If such a match can be found, the corresponding production is applied to the host graph. These steps are iteratively performed until no production in R fulfills the condition in lines 8 and 9.

```
algorithm MapSchema(R, S)
1)    input R, a set of forward and reverse productions derived from a triple graph grammar
2)    input S, a start graph (according to Figure 14)
3)    output G, a migration graph (according to Figure 8)
4)    begin
5)        let G=S
6)        repeat
7)            let r:(P,Q,C,T)∈R be a production that fulfills the following conditions
8)                - P has a match in G represented by a morphism m:P→G
9)                - this match cannot be extended in G by a match for the SMG elements in Q
10)           let G = G↓(r,m)
11)       until no production p∈P fulfills the conditions in lines 8 and 9
12)   return G
13)   end
```

Fig. 15. Algorithm MapSchema.

The described algorithm defines how triple graph grammars can be employed for bi-directional schema translation. Note, that the productions are not tested and applied in a pre-defined order. (The specification of the schema mapping rules ensures *confluence* [40, p. 105] for all production applications.) For larger schemas this simple algorithm lacks efficiency. This problem can be solved by implementing a procedural framework that defines an order for the application of the derived graph productions. The procedural framework that has been implemented in our DBRE environment is described in [23].

The main advantage of using triple graph grammars to specify and implement schema translators is their high level of abstraction. Graph-oriented specifications are much easier to define, comprehend, and extend than program code and textual formalisms. Another benefit of this approach is that it enables the generation of bi-directional translators, because it defines correspondences between increments in both data models.

6. Schema redesign-transformations

In the previous section, we described and specified a canonical translation from an analyzed logical schema to a conceptual schema (and vice versa). Even though the presented mapping rules define a bi-directional mapping between logical and conceptual schemas, it is important to note that this mapping is partial: further mapping rules are needed to define correspondences between additional conceptual constructs like aggregations and *many-to-many* relationships. These mapping rules can be defined analogously to the rules described before (cf. [53]). Typically, their definition leads to ambiguities in the reverse translation process from the logical to the conceptual schema. For example, a given R-IND can be mapped to an association or an aggregation, an RS with two foreign keys can be mapped to a class or to a *many-to-many* relationship, etc. Such ambiguities can be solved by adding priorities to mapping rules [26] or extending the logical schema by further semantic annotations, e.g., to mark an

aggregation relationship. Still, we made the experience that the number of mapping rules grows very large if we strive to consider all possible (and reasonable) correspondences between logical and conceptual schema constructs. This large number of mapping rules soon became incomprehensible for the reengineer.

We tackle this problem by combining a limited set of mapping rules with a set of conceptual *redesign transformations*. The reengineer can use these redesign transformations to choose from alternative conceptual constructs while the correspondences to the logical schema are kept automatically. In addition, such redesign transformations may be used to extend or modify the conceptual schema in order to meet new requirements (cf. Fig. 3).

Redesign transformations have traditionally been applied in logical database design [4, p. 424]. For example, they are used as decomposition operations in algorithms to obtain a normalized relational database schema. Several researchers have proposed catalogs of redesign transformations for conceptual schemas, e.g., [3,18,43,50,7]. Typically, these catalogs consist of the so-called *primitive* transformations which serve as the basic building blocks of more *complex* transformations. Banerjee et al. argue that their catalog of transformations is complete [3]. Still, Schiefer shows examples for important schema transformations that cannot be performed with this catalog [43, p. 54]. Especially, in the context of DBRE, we doubt the feasibility of defining a complete catalog of schema redesign transformations. This is because legacy database schemas often comprise complex idiosyncratic optimization patterns and unforeseen design structures [6]. In most cases, it is not sufficient to apply primitive transformations to the building blocks of a complex optimization pattern but the entire pattern has to be converted as such. Hence, our special focus is on providing a catalog of transformations that is easily *extensible* rather than trying to create a catalog that is *complete*. The combination of the expressive power of graph grammar productions with the *Progres* code generation mechanism [44] enables us to achieve this goal: the catalog of redesign transformations that are provided by our schema migration tool can easily be extended and customized on a high level of abstraction.

Many approaches in the domain of database evolution allow for the reorganization of the data after a redesign transformation has been applied to the schema [43,50]. In our application, we focus on integrating legacy database schemas with distributed, object-oriented technology by generating a middleware component that provides data integration. The necessary schema dependency information is represented by the SMG (cf. Section 4). Consequently, our redesign transformations update the SMG in correspondence to the structural transformation of the conceptual schema. The mapping information provided by the updated SMG allows us to use the redesigned conceptual schema as an object-oriented view on the implemented logical schema and to achieve a gradual migration.

Our current catalog of redesign transformations comprises about 30 transformations. See Fig. 3 for several application examples. In this section, we exemplify the specification of the redesign transformation *SplitClass* in form of graph productions.

Redesign transformations are performed interactively by the reengineer who provides the parameters included in the signature of the graph production. Transformation *SplitClass* creates a new class with name *clName* which is connected by a total *one-to-one*

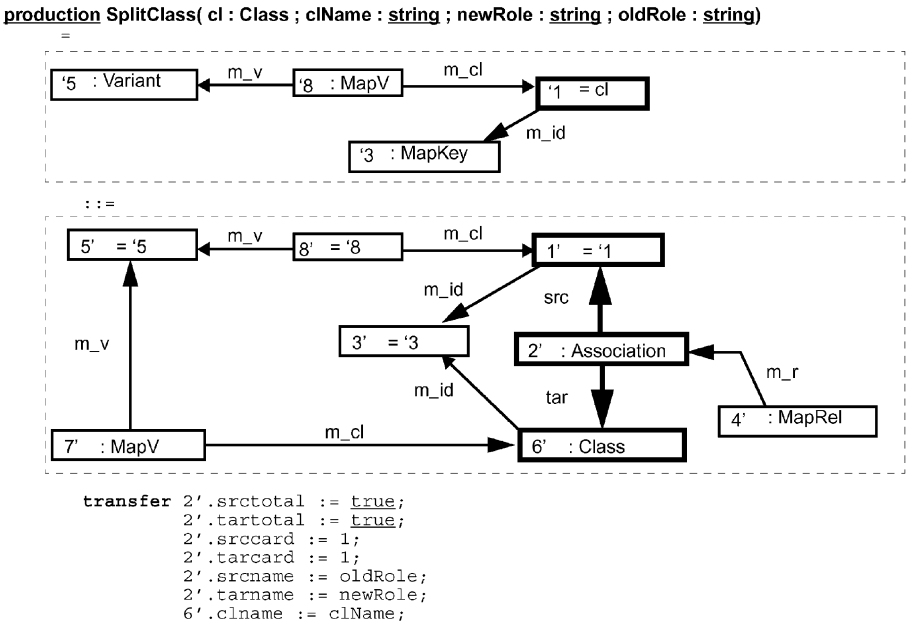

Fig. 16. Schema transformation *SplitClass*.

association to a given class *cl* (cf. Fig. 16). Parameters *oldRole* and *newRole* contain the role names of the pre-existing class and the new class in the created association, respectively. In Fig. 16 we use bold nodes and edges to make it easier to distinguish the part of the production that specifies the actual change in the conceptual schema from the remaining part that specifies the modification in the mapping graph. Production *SplitClass* specifies that the newly created class (Node 6′) is mapped (by Node 7′) to the same variant (and RS) that has been mapped to the pre-existing class (Node 1′). A new edge of type *m_id* represents the information that OIDs of the new class are translated to the same value-based key like OIDs of the old class. The new association is not mapped to any foreign key (R-IND) in the relational schema. However, it is connected to a new node of type *MapRel* to indicate that the association already has a corresponding representation in the logical schema (cf. the mapping algorithm in Fig. 15).

Classes that are newly created by applying transformation *SplitClass* do not contain attributes or participate in any relationship other than the newly created association. The reengineer can use transformations like *CreateAttribute* or *CreateAssociation* to create new class properties. In this case, the mapping rules defined in Section 5 are used to translate these new properties to columns and foreign keys which extend the original logical schema. Besides the possibility to add new properties, we provide two transformations, *MoveAttribute* and *MoveAssociation*, that allow for the moving of class properties from one class over a *one-to-one* association to another class. These transformations do not augment the information capacity of the schema. Hence, they

do not imply changes in its implementation. The graph production for transformation *MoveAttribute* is presented in Appendix C.

Basic redesign transformations like *SplitClass* and *MoveAttribute* may be combined to more powerful redesign transformations using the so-called *graph transactions* in *Progres*. For example, we have implemented another version of transformation *SplitClass* that accepts a set of attributes as additional parameter. This complex version of *SplitClass* first calls the basic version of *SplitClass* creating an association to an empty new class. Subsequently, it loops through the set of attributes passed as parameter and moves them to the new class using *MoveAttribute*. Using this complex *SplitClass* transformation, the user is unburdened from the tedious task of applying *MoveAttribute* manually several times. Note, that *Progres* graph transactions are executed as atomic operations i.e., with an "*all-or-nothing*" semantics.

Splitclass and *MoveAttribute* are examples for redesign transformations that maintain consistency with the corresponding logical schema directly. They do not require changes to the logical schema and no change of the legacy database is necessary. In the database literature, schema redesign transformations include a definition of the *semantics* of the specified schema change. This semantics is declared by a definition of how instances of the source schema are translated to instances of the target schema. Hence, a redesign transformation is often defined as a tuple (T,I), where T denotes the so-called *structure transformation* and I is the *instance mapping* [18]. In this paper, we do not specify the instance mapping of redesign transformations explicitly, but we describe their semantics by defining the modification to the SMG in correspondence to the structural transformation of the conceptual schema. The rationale for this approach is that we focus on integrating legacy DB schemas with distributed, object-oriented technology by generating a middleware component that provides data integration. The schema dependency information necessary for this integration is represented by the SMG. Using a data integration middleware, the conceptual schema represents an object-oriented view on the implemented logical schema. Redesign transformations that are performed to this view do not necessarily change the implemented data model. In fact, we are interested in keeping the modifications of the legacy schema to a minimum to preserve compatibility with existing legacy application code and data. Only transformations that extend the information capacity of the conceptual schema require actual changes in its implementation. This means that in our example, transformations *Rename*, *Generalize*, *AggrInTuple*, *SplitClass*, *RemoveArtKey*, and *ClassToOrderedAssoc* do *not* imply changes in the logical schema because they preserve the *information capacity* [4] of the schema (Fig. 3 on p. 5). In [30], we employ the theory of parallel graph transformation systems to formalize specifications of structure transformations *and* instance mappings and prove their property of information capacity preservation.

7. Incremental change propagation

Inconsistencies between different representations on various levels of abstraction often cause update problems in reengineering projects. Whenever the reengineer discovers new information about the real semantics of implementation constructs in the

legacy schema, its conceptual representation that has been created so far must be updated accordingly. A further typical source of inconsistencies are *on-the-fly* modifications to the implementation of the legacy database due to urgent requirements while the reengineering project is in progress. Detecting and eliminating such inconsistencies manually is a time-consuming and error-prone activity. Hence, a commonly used alternative is to discard all created conceptual views of the legacy database and generate default representations anew. In this case, the redesign work that has been performed manually by the reengineer is lost and has to be repeated. Obviously, both alternatives are unsatisfactory. Therefore, we have developed an *incremental* approach to change propagation in database reengineering environments. In this section, we describe an automatic mechanism to propagate changes of an legacy database's implementation to its conceptual representation without discarding manually performed redesign operations that remain valid.

The developed consistency management mechanism is based on the fact that our approach to schema migration employs transformations as the fundamental concept. In Section 5, we have shown how to derive an automatic transformation system from a triple graph grammar to translate a logical database schema into an initial conceptual representation. In Section 6, we have proposed to use redesign transformations that can be applied to this conceptual representation, interactively. The main idea of our consistency management concept is to keep track of input/output dependencies between all transformations that have been applied to the original legacy schema. In the case of implementation changes or modified semantic annotations, this dependency information is employed to detect all transformations which are affected by the change. Each of these transformations is reevaluated automatically to determine if its pre-conditions are still valid. Only those transformations which have lost their applicability are discarded.

We have used graph productions to formalize and implement transformations. In this sense, the left-hand side of a graph production represents the input of the corresponding transformation, while its output is represented by the right-hand side. If we want to maintain input/output dependencies of applied transformations, we have to store information about the matches for the corresponding graph productions. A graph-based structure is most suitable to maintain these dependencies. We call the corresponding graph *history graph* because it reflects the migration history of a legacy database schema. Fig. 17 illustrates the basic structure of a history graph: applied transformations are explicitly represented by *T*-nodes with corresponding input and output parameters. Input parameters which have actually been removed by an applied transformation remain as place holders in the history graph to represent the necessary dependency information (cf. *C*-nodes with gray shape in Fig. 17).

In order to maintain the application contexts of transformations we have to identify and represent the graph elements on their left- and right-hand sides explicitly in the history graph. For example, let us consider transformation *SplitClass* in Fig. 16. It has four input node parameters and eight output node parameters. Each parameter has a unique node number and some of the output parameters also serve as input. Fig. 18 shows this input/output structure for transformation *SplitClass*. The parameter nodes serve as place holders for the actual parameters of a transformation application. Hence, we call this structure a transformation template.

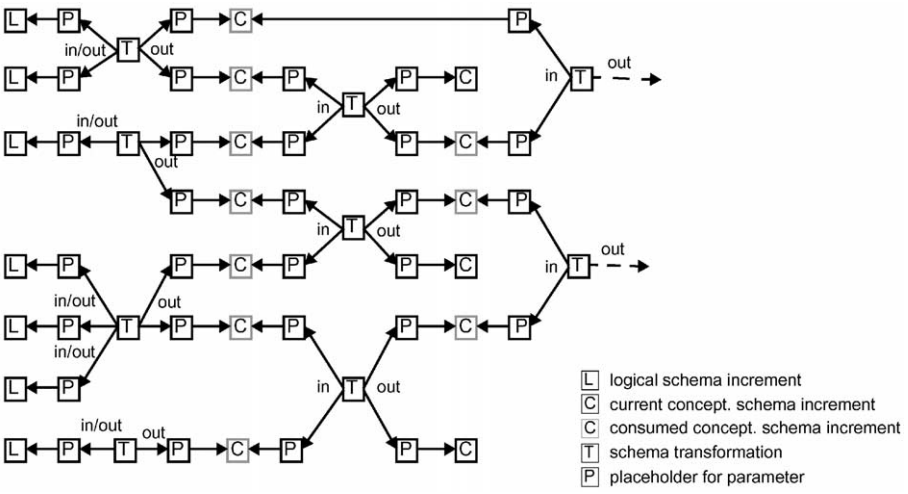

Fig. 17. Basic structure of a history graph.

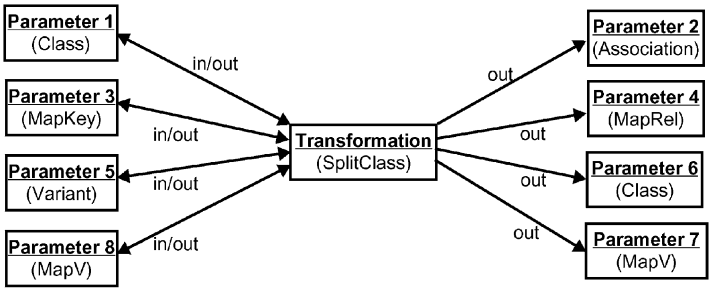

Fig. 18. Template of transformation *SplitClass*.

The general history graph in Fig. 17 shows distinct input and output parameters for each transformation. We chose this representation to simplify the layout and improve the comprehension of our approach. Still, Fig. 18 shows that some input and output parameters might be identical for certain transformations (cf. in/out edges). Fig. 19 illustrates this situation with our practical application example from Section 2. Note that in this concrete graph, we have skipped the P-nodes for the sake of simplicity.

Even though node parameters are sufficient to determine the application context of a *Progres* production, Fig. 19 shows that the transformation itself depends also on edge parameters. In *Progres*, these dependencies cannot be represented directly in the history graph because the underlying graph model does not allow for higher-order edges, i.e., edges that have edges as their source or target (cf. Definition 1). However, this dependency information can be disregarded if all graph productions comply to the requirement that whenever an edge is modified its source and target nodes have

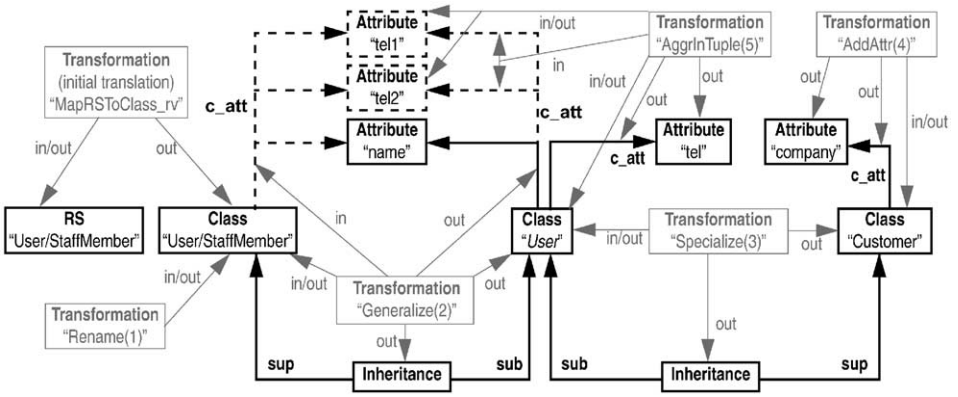

Fig. 19. Concrete history graph (cut-out).

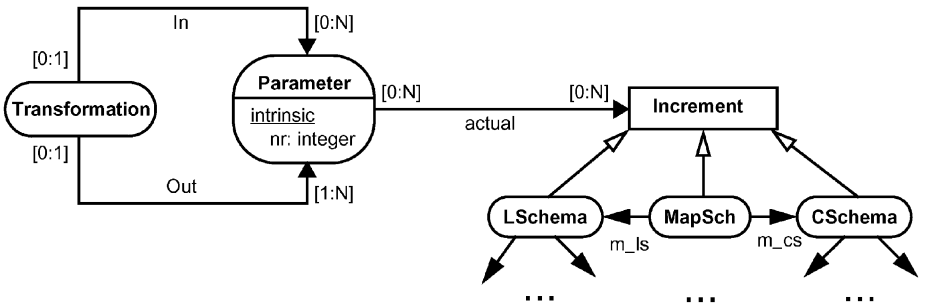

Fig. 20. History graph model.

to occur on the left-hand sides. This requirement is satisfied in all graph productions included in our environment.[4]

Fig. 20 shows a graphical *Progres* specification for the history graph model. Input and output dependencies are represented by edges of type *In* and *Out*. Fig. 20 also shows that the history graph model is an extension of the migration graph model, i.e., the history graph contains the migration graph as a subgraph. Node type *Increment* represents a generalization of all node types in the migration graph model represented in Fig. 8. Edges of type *actual* connect parameter place holders of transformation templates with their actual input and output parameters in the migration graph.

Definition 3. *History graph*[5]: The *history graph* is a graph that includes the migration graph as a subgraph. Moreover, it contains nodes and edges that represent all

[4] *Progres* provides additional specification concepts like *path expressions* (cf. Section 5.2), *negative* application conditions, and *optional* nodes. For the sake of simplicity, we will not discuss the representation of the dependencies induced by these additional concepts in the history graph but refer to [25] instead.

[5] The history graph defined above is a specific implementation of the general concept of a *graph process* as introduced in [11]. A graph process is a partially ordered structure, plus suitable mappings which relate the elements of this structure to those of a given typed graph grammar.

application contexts of (mapping and redesign) productions in the entire editing history. The corresponding extension of the migration graph model (cf. Fig. 8) is given in Fig. 20. The projection of a history graph $H : (N, E, t_N, A)$ on the current migration graph $MG(H) : (N', E', t'_N, A')$ includes all increments which do not occur as in-parameters of a transformation without occurring as out-parameters of the same transformation, i.e.,

- $N' := \{n \in N \mid t_N(n) \notin \{\text{“Transformation”},\text{“Parameter”}\} \wedge \forall n_p, n_t \in N, \forall e_a, e_i \in E : t(e_a) = n \wedge s(e_a) = n_p \wedge t(e_i) = n_p \wedge s(e_i) = n_t \wedge t_E(e_a) = \text{“actual”} \wedge t_E(e_i) = \text{“In”} \Rightarrow \exists e_o \in E : t(e_o) = n_p \wedge s(e_o) = n_t \wedge t_E(e_o) = \text{“out”})\}$
- $E' := \{e \in E \mid s(e), t(e) \in N'\}$
- $t'_N := t_N \backslash \{\text{‘Transformation’, ‘Parameter’}\}$
- $A' = A$

In order to log the application of transformations in the history graph we have to redefine the way how transformations (graph productions) are applied (cf. Definition 4 in Appendix A). The main difference is that nodes which are deleted on the right-hand side of a production are not removed from the history graph but they are isolated, i.e., all their in- and out-going edges in the corresponding migration graph are deleted.

In the rest of this section, we describe how the information stored in the history graph can be used for incremental change propagation. Let us revisit the scenario from Section 2 in which an analyzed logical database schema had been translated to a conceptual representation which subsequently was redesigned and extended. Our case study describes a sample situation for a change in the logical schema during such an ongoing conceptual migration process (cf. Fig. 5). Using the history graph that has been created during the translation and editing history, the change propagation process has four major phases, namely *forward propagation*, *backward propagation*, *reevaluation*, and *translation*.

7.1. Forward and backward propagation

In the first phase forward propagation, the input/output dependencies in the history graph are used to detect all transformation applications (and increments in the conceptual schema) which are affected by the modifications in the logical schema. This step is illustrated in Fig. 21 where L-nodes marked with a pencil icon represent modifications and extension of the logical schema, respectively.

Obviously, all transformation applications that have been marked in the forward propagation step have to be validated. However, some of these transformation applications depend on input parameters which were consumed by a transformation. These parameters, which are only represented by isolated place holders, have to be reproduced before the dependent transformation can be reevaluated. Reproducing these parameters means to reevaluate all transformations that have been applied to produce them. Some of the transformation applications that have to be reevaluated might not have been marked in the forward propagation phase because they are not directly affected by the modification in the logical schema. Hence, we need a further *backward propagation*

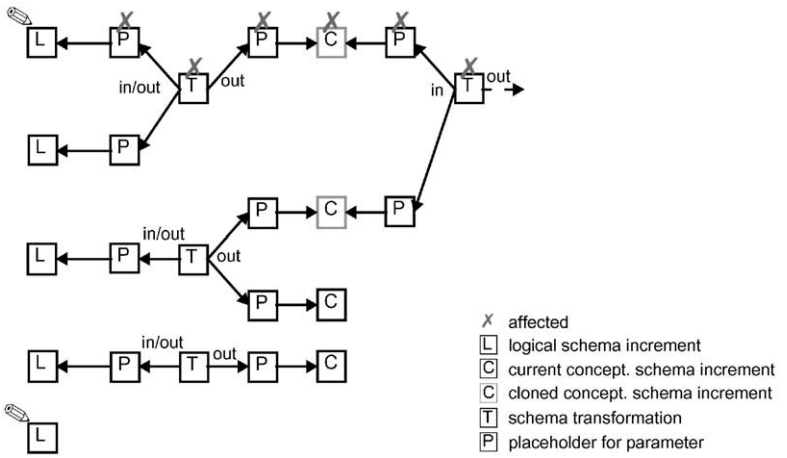

Fig. 21. Phase I: forward propagation.

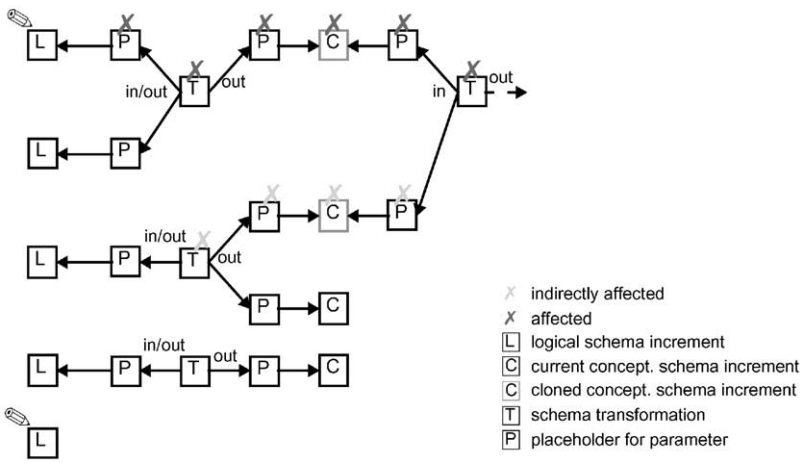

Fig. 22. Phase II: backward propagation.

phase to mark such indirectly affected transformation applications in the history graph (cf. Fig. 22).

7.2. Reevaluation

In the third phase *reevaluation*, the marked transformation applications are re-evaluated in the pre-defined order of their input/output dependencies. Reevaluating a transformation application means to apply the corresponding transformation anew to the current (maybe changed) parameters. Each transformation that remains applicable remains in the history graph. Fig. 23 shows that the output parameters of such a

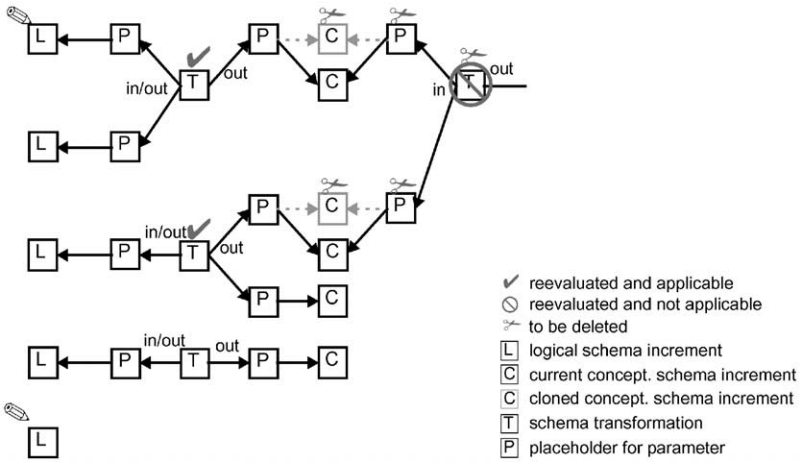

Fig. 23. Phase III: reevaluation.

transformation and the input parameters of a dependent transformation application are updated to the newly created conceptual schema increments. All old parameter place holders are deleted from the history graph. Likewise, all transformations which are no longer applicable are deleted as well. In Fig. 23, this is illustrated for the right-most transformation template.

7.3. Translation

The purpose of the final phase, *translation,* in the change propagation process is to translate logical schema increments which do not have a current representation in the conceptual schema (cf. Fig. 24). This is necessary for logical schema increments which have been added during the last modification. Furthermore, translations of existing logical schema increments might have been deleted during the reevaluation phase because the corresponding transformation rules are no longer applicable. At the end of this translation phase, the consistency of the logical schema with its conceptual representation has been reestablished.

The described incremental change propagation algorithm has been implemented in *Varlet* using the *Progres* language and environment. This implementation is described in detail in [53]. Fig. 25 shows the transaction *PropagateChange* which formalizes the propagation process. It requires an argument *changeSet* which represents the set of all logical schema increments that have been added or modified. (These increments can easily be collected by a CARE tool (like the Varlet Analyst) during interactive schema analysis activities.) In the first phase, path expressions are used to collect all directly affected transformation applications in the local variable *affectedTrafoAppls*. Note, that a path expression with a supplemented asterisk symbol (*) computes the transitive closure of this path. In the backward propagation phase all transformation applications are added to variable *affectedTrafoAppls* which are needed to reproduce consumed parameters. Phase III is performed in a loop that repeatedly chooses one

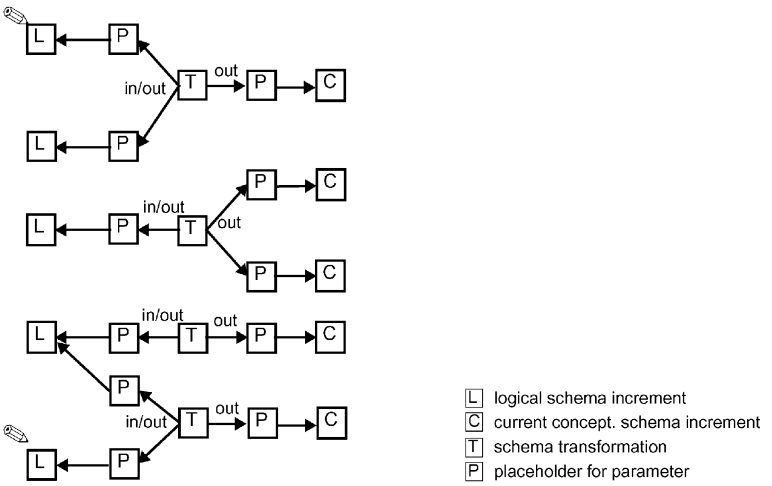

Fig. 24. Phase IV: translation.

transformation application (*oldTrafoAppl*) that does not depend on any other transformation application in *affectedTrafoAppls*. Note, that the *Progres* operator *and* computes the intersection of two sets. The following *choose* statement tries to reapply the transformation in *oldTrafoAppl*. If this is possible and the specified invariant graph constraints are fulfilled it actualizes the output parameters of the new transformation application. (An example for invariant graph constraint is the UML class naming convention, two classes are not allowed to have the same name.) Subsequently, the reevaluated transformation application *oldTrafoAppl* is removed from the set *affectedTrafoAppls*. This is done by using the *Progres* operator *but_not* which computes the difference of two sets. In the case that the transformation in *oldTrafoAppl* has lost its applicability, the *else* block of the *choose* statement in Fig. 25 collects all dependent transformation applications in variable *depTrafoAppls*. Subsequently, these transformation applications are removed from the history graph.

8. Conclusion

Providing tools that allow for an iterative and explorative reengineering process is a challenging but important goal of current research. The approach presented in this paper is one step in this direction in the domain of reengineering legacy database schemas. A formal specification of all operations applied to a legacy schema allows for the propagation of modifications in case of iterations between analysis and redesign activities. The technique described in this paper can be used analogously to improve the usability of many existing transformation-based software refactoring tools. The rationale behind our decision to describe our transformations with graph rewriting rules is that graph grammars are executable *and* intuitively well understandable. Moreover, this choice enabled us to use the programmed graph rewriting systems (*Progres*)

```
transaction PropagateChange( changeSet : Increment [1:n]) =
   use
     affectedTrafoAppls, depTrafoAppls : Transformation [0:n];
     oldTrafoAppl, newTrafoAppl : Transformation
   do

(* Phase I: forward propagation *)

       affectedTrafoAppls := changeSet.(    <-actual-
                                        &  <-In-      )
     & affectedTrafoAppls :=
         affectedTrafoAppls.( (    -Out->
                               &  -actual->
                               &  <-actual-
                               &  <-In-      ) * )

(* Phase II: backward propagation *)

& affectedTrafoAppls :=
         affectedTrafoAppls.( (    -In->
                               &  -actual->
                               &  <-actual-
                               &  <-Out-     ) * )

(* Phase III: reevaluation *)

     & loop
        oldTrafoAppl :=
          affectedTrafoAppls.valid (empty ( (self.-In->.-actual->.<-actual-.<-Out-)
                                            and affectedTrafoAppls
        & choose
            Reevaluate ( oldTrafoAppl, out newTrafoAppl )
          & CheckGraphConstraints
          & ActualizeOutParams ( oldTrafoAppl, newTrafoAppl )
          & affectedTrafoAppls :=
              (affectedTrafoAppls but not oldTrafoAppl)
          else
            depTrafoAppls :=
              oldTrafoAppl.( (    (          -Out->
                                   but not  -In->  )
                               &  -actual->
                               &  <-actual-
                               &  <-In-               ) * )
          & RemoveTrafoAppls ( depTrafoAppls )
          & affectedTrafoAppls :=
              (affectedTrafoAppls but not depTrafoAppls)
          end
      end

(* Phase IV: translation *)

     & MapSchema (* cf. Figure 15 *)

   end
 end;
```

Fig. 25. Transaction *PropagateChange*.

environment to rapidly prototype and evaluate the described technique within *Varlet*. With *Progres*, executable code can be automatically generated from specified graph productions. Thus, the *Varlet* environment facilitates customization of (redesign) operations by simply adding new or changing existing graph rewriting rules. The current (graphical) *Progres* specification consists of 300 pages. From this specification we generate 180,000 lines of C code, which implements the core component of *Varlet*. The user interface is implemented using TCL/TK.

The *Varlet* environment has been tested and refined in the context of an industrial project in collaboration with two German companies. The analyzed logical schema

included 85 tables, 347 attributes, and 138 INDs. The automatic initial translation to the conceptual data model took 2.5 min on a SUN Ultra-Sparc II with 300 Mhz. In experiments with several (internal and external) users, we have validated the usefulness of the proposed automatic change propagation mechanism to support process iterations. The most frequent changes of the logical schema have been due to additional INDs or changed semantic classifications of INDs. Depending on how many applied redesign transformations have been affected by a given change the propagation time ranged from 30 s up to minutes. The users considered this performance as satisfactory compared to the tedious and error-prone alternative of validating and reestablishing the consistency manually. The *Varlet* prototype is available for academic purposes at *http://varlet.uvic.ca*.

Besides the aforementioned positive results, there exist a number of problems with our current approach that we would like to address in the future. One problem is that using *Progres* as our development platform impedes the portability of our generated reengineering tools. The *Progres* system (and all generated tools) depends on a non-standard database system (GRAS) that is available only on very specific UNIX platforms. In addition to this portability issue, the requirement for GRAS results in a fairly large footprint of our generated tools. Moreover, GRAS can only store a maximum number of 64,000 objects. This limit becomes a problem for larger legacy systems, because GRAS objects are used to store both schema representations (logical and conceptual) as well as the entire editing history of the reengineering project (in terms of the history graph).

Another problem concerns the current design of our consistency management component: in the current design, the consistency management component is tightly coupled to a graph-based representation of the legacy system artifacts. In fact, Fig. 20 shows that we use inheritance to couple the history graph to the data structures representing the legacy system artifacts. This deep integration is suitable for the development of new graph-based reengineering tools. It is less suitable, however, for integrating a consistency management component into existing environments. In these situations, it would be desirable to maintain the history graph and the change propagation algorithm in a separate consistency management component that can be plugged into reengineering tools using the existing API and scripting functionality of these tools.

Our current and future research aims are to develop such a generic and reusable consistency management component for reengineering tools. The most important problem to be solved in this context is to determine the minimal functionality required from a reengineering tool in order to be able to integrate it with a consistency management component. The implementation will be based on the platform-independent programming language Java and the graph grammar engineering environment FUJABA [15].

Acknowledgements

We would like to thank eps Bertelsmann, Gütersloh, Germany, for providing us with an industrial-strength case study for our research prototype. Thanks to the Progres Team

at RWTH Aachen, Germany, for their technical support. We thank Derek Church for editing the language of this publication. Finally, we thank the anonymous reviewers of the previous version of this paper for their valuable comments and suggestions. The described research has been supported in part by the National Science and Research Counsel of Canada (NSERC).

Appendix A. Application of a production

Definition 4. *Application of a production*: A production $r : (P, Q, C, T)$ is *applied* to graph G in the following five steps:

- *CHOOSE* an *occurrence* of the left-hand side P in G. P has an occurrence in graph G if there is a morphism $m : P \to G$ which preserves source and target and labeling mappings. Furthermore, the occurrence has to fulfill the so-called identification condition which prescribes that elements on the left-hand side which do not occur on the right-hand side can uniquely be identified in G, i.e., $\forall x \in P \backslash Q, x' \in P : m(x) = m(x') \Rightarrow x = x'$.
- *CHECK* the application conditions according to C. If they are fulfilled the occurrence of P in G is called a *match* for P.
- *REMOVE* all elements in G which have been matched to elements in P that do not occur in Q, i.e., remove $m(P \backslash Q)$ from G. If the removal of nodes causes dangling edges in G these dangling edges are removed as well.
- *ADD* all elements to G which are new in Q, i.e., which do not occur in P. These new elements are glued to G in the preserved graph elements identified by $m(P) \cap Q$). We denote the morphism $\bar{m} : Q \to G$ that identifies the (newly created) occurrence of Q in G as *comatch*.
- *TRANSFER* attribute values to nodes in G that match nodes in Q according to the attribute transfer clauses specified in T.

Appendix B. Examples for mapping rules

In the relational data model, the representation of logical entities and their relationships is based on the simple mathematical concept of relations. Hence, columns are basically used for two purposes: they might represent actual data values of entities or they might represent references implemented as redundant copies of such data values in other relations (foreign keys). Only columns that do not represent foreign keys should be mapped to attributes in the conceptual model because it includes explicit concepts for relationships (associations and aggregations). This restriction is considered within the first part of the reverse application condition of mapping rule *MapColToAttr* (cf. the comment in Fig. 26).

Even though an RS with multiple variants is mapped to an inheritance hierarchy of classes, each of its columns is mapped to only one class attribute in this hierarchy. This attribute is then inherited by all subclasses in the hierarchy (cf. Fig. 9). The second part of the reverse application condition ensures that the column is mapped to

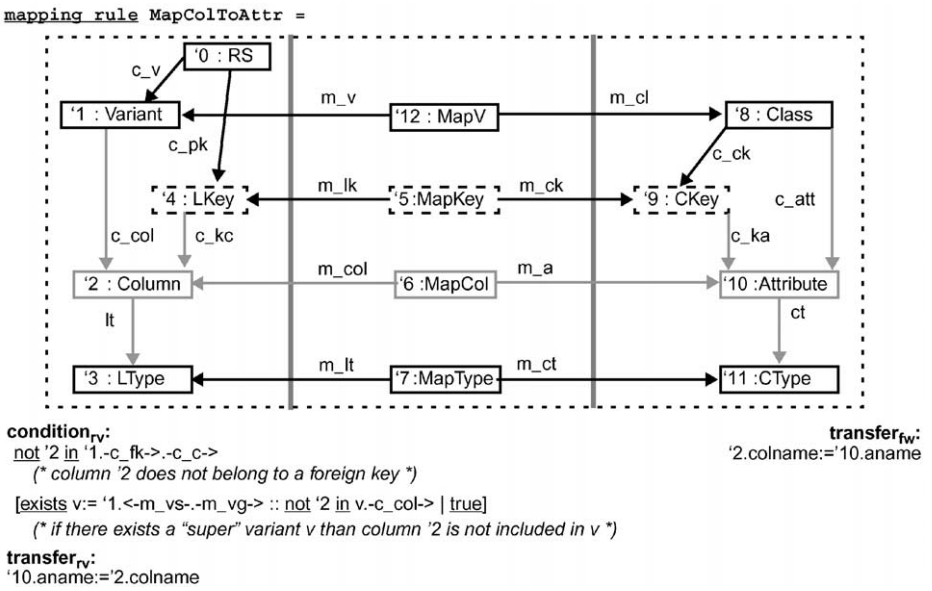

Fig. 26. Mapping rule *MapColToAttr*.

the most general class (′8) in the inheritance hierarchy. This requirement is represented by a *conditional boolean expression* [49, p.44] which returns *true* if there exists no such generalization. Otherwise, it ensures that the variant that has been mapped to the generalization of class ′8 does not include column ′2. Note, that the operator *in* tests the membership of its first argument in the set represented by its second argument.

Nodes ′4, ′5, and ′9 have been declared as *optional* graph elements, indicated by dashed boxes, cf. [49]. The optional graph elements deal with the two possible cases of mapping key columns or non-key columns. If the column (respectively the attribute) belongs to a key this information is reflected by adding the corresponding syntactical edges in both ASGs. Otherwise, nodes ′4, ′5, ′9 and the corresponding edges are ignored.

In contrast to the variety of concepts for relationships in the conceptual model (inheritance, association, and aggregation with different cardinalities), INDs are the only means to implement references between different RS in the relational model. During the schema analysis phase, we aim to narrow this semantical gap by classifying INDs either as normal references with different cardinalities (R-IND) or as inheritance relationships (I-IND). Based on this classification, we employ different mapping rules that translate INDs to relationships in the conceptual model and vice versa.

Rule *MapRINDToAssoc[1:1]* in Fig. 27 maps an R-IND which is inversely key-based to a total *one-to-one* association in the conceptual model. The restriction to inversely key-based INDs with an inverse IND is specified by testing attribute *invkb* and *inverse* in the textual condition part of rule *MapRINDToAssoc[1:1]*. In analogy to the previous mapping rules, the rest of this condition block ensures that the new

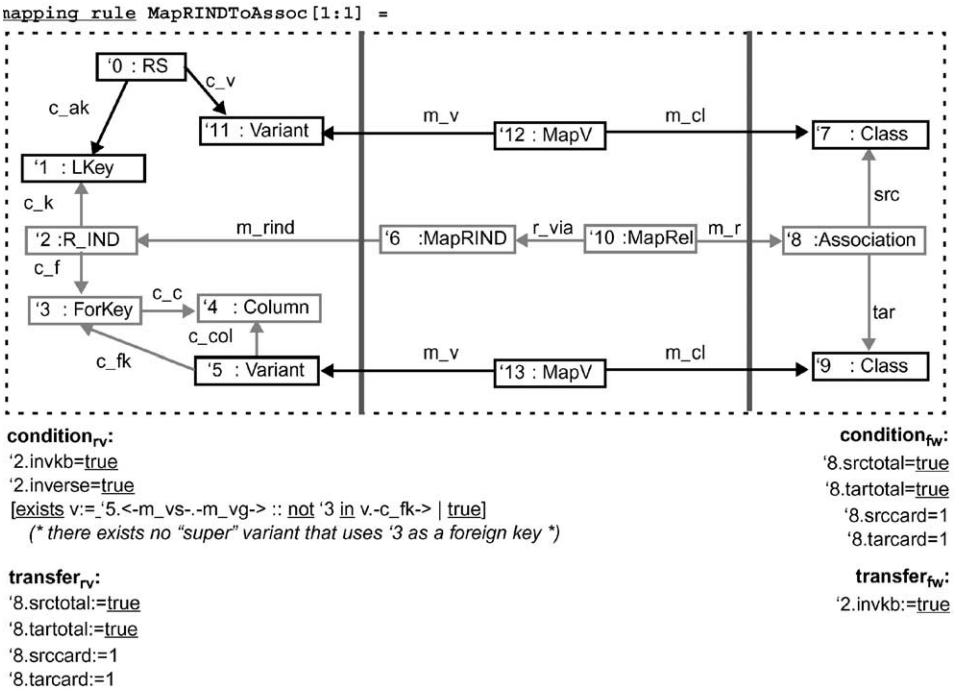

Fig. 27. Mapping rule *MapRINDToAssoc*[*1* : *1*].

association is created between the most general classes in the corresponding inheritance hierarchy.

Appendix C. Schema redesign transformation MoveAttribute

Like in Fig. 16, we use bold nodes and edges to distinguish the part of the production that specifies the actual change in the conceptual schema from the modification in the mapping graph. In Fig. 28, the two parameters *attr* and *assoc* represent the attribute that has to be moved and the association that connects source and target of this relocation operation. The right-hand side of production *MoveAttribute* shows that the attribute which was initially aggregated in class $'1$ by a *c_att* edge has been relocated to class $3'$ after the transformation has been applied. The information about the relocation is reflected in the mapping graph by adding the *MapRIND* node $'5$ to the access path of the relocated attribute which have been mapped to the association. This is done by adding an *a_via* edge from the attribute mapping node $'6$ to node $'5$. Still, it is also possible that association *assoc* is not mapped to a *MapRIND* node, e.g., if it has been created by applying the *SplitClass* transformation. Hence, node $'5$ is defined to be optional.

The application condition of production *MoveAttribute* restricts its applicability to *one-to-one* associations, only. The relocation of class properties over *many-to-one*

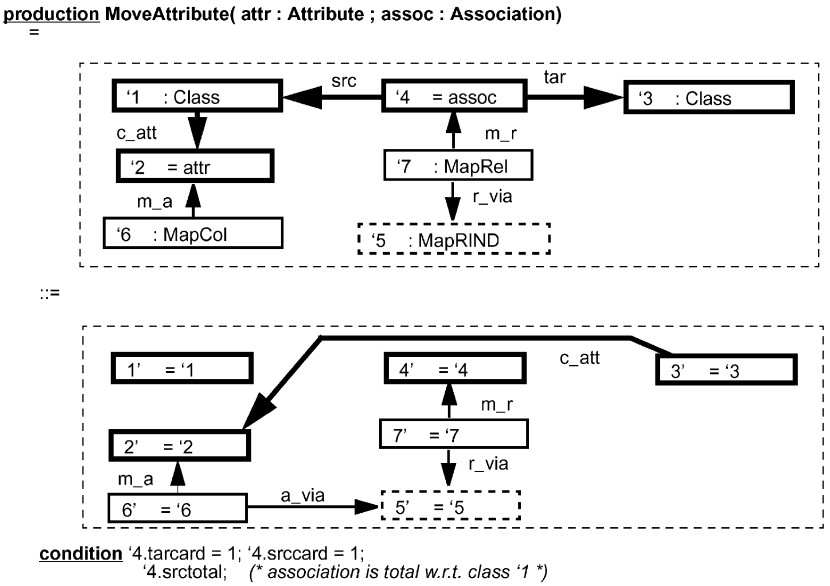

Fig. 28. Schema transformation Moveattribute.

associations is ambiguous w.r.t. to the instance conversion and, thus, has to be prohibited. In analogy, association ′4 has to be total w.r.t. class ′1. On the other hand, relocating class properties over a *one-to-many* association would imply changes to the underlying logical schema. In the case that a relocation operation intends such a change, the corresponding properties have to be deleted from the variants mapped to class ′1 and added to the variants mapped to class ′3. This can be done by a concatenation of *remove* and *create* transformations. Again, the mapping rules of Section 5 are used to propagate such changes to the logical schema. Strategies to reorganize the available data after such changes have been developed in the domain of database evolution [43,50]. One typical solution is to insert default values for undefined attribute values.

References

[1] P. Aiken, Data Reverse Engineering: Slaying the Legacy Dragon, McGraw-Hill, New York, 1995.
[2] M. Andersson, Extracting an entity relationship schema from a relational database through reverse engineering, in: Proc. of the 13th Intl. Conf. of the Entity Relationship Approach, Manchester, Lecture Notes in Computer Science, vol. 881, Springer, Berlin, 1994, pp. 403–419.
[3] J. Banerjee, W. Kim, H.-J. Kim, H.F. Korth, Semantics and implementation of schema evolution in object-oriented databases, SIGMOD Record (Proc. Conf. on Management of Data) 16 (3) (1987) 311–322.
[4] C. Batini, S.Ceri, S.B. Navathe, Conceptual Database Design, Benjamin/Cummings, Menlo Park, CA, 1992.
[5] A. Behm, A. Geppert, K.R. Dittrich. On the migration of relational schemas and data to object-oriented database systems, in: Proc. 5th Internat. Conf. on Re-Technologies for Information Systems, Klagenfurt, Austria, Österreichische Computer Gesellschaft, December 1997, pp. 13–33.

[6] M. Blaha, W. Premerlani, Observed idiosyncrasies of relational database designs, In: Second Working Conf. on Reverse Engineering, Toronto, Ontario, Canada, IEEE CS Press, Silver Spring, MD, 1995.
[7] M. Blaha, W. Premerlani, A catalog of object model transformations, in: Proc. of 3rd Working Conference on Reverse Engineering, Monterey, CA, USA, November 1996.
[8] M. Blaha, W. Premerlani, Object-Oriented Modeling and Design for Database Applications, Prentice-Hall, Englewood Cliffs, NJ, 1998.
[9] H. Briand, C. Ducateau, Y. Hebrail, D. Herin-Aime, J. Kouloumdjian, From minimal cover to entity-relationship diagram, in: Proc. of the 6th Intl. Conf. of the Entity Relationship Approach, New York, North-Holland, Amsterdam, November 1987, pp. 287–304.
[10] R.G.G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Staniendam, F. Velez, The Object Data Standard: ODMG 3.0, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2000.
[11] A. Corradini, U. Montanari, F. Rossi, Graph processes, Fundamenta Informaticae, IOS Press, Amsterdam, vol. 26(3), June 1996, pp. 241–265.
[12] G. Engels, Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung, Ph.D. Thesis, Universität Osnabrück, 1986, VDI-Verlag.
[13] V. Englebert, J.-L. Hainaut, DB-MAIN: a next generation meta-CASE, J. Inform. Systems—Special Issue Meta-CASEs 24 (2) (1999) 99–112.
[14] C. Fahrner, G. Vossen, Transforming relational database schemas into object-oriented schemas according to ODMG-93, In: Proc. of the 4th Intl. Conf. on Deductive and Object-Oriented Databases, Lecture Notes in Computer Science, vol. 1013, Springer, Berlin, 1995.
[15] T. Fischer, J. Niere, L. Torunski, A. Zündorf, Story diagrams: a new graph rewrite language based on the unified modeling language, In: G. Engels, G. Rozenberg (Eds.), Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany, Springer, Berlin, 1998.
[16] J. Fong, Converting relational to object-oriented databases, ACM SIGMOD Record 26(1) (1997) 53–58.
[17] J.-L. Hainaut, A generic entity-relationship model, in: Falkenberg, Lindgren (Eds.), Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: An In-depth Analysis, North-Holland, Amsterdam, 1989.
[18] J.-L. Hainaut, Entity-generating schema transformations for entity-relationship models, in: Proc. of the 10th Entity-Relationship Conference, San Mateo, 1991.
[19] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland, Requirements for information system reverse engineering support, Technical Report RP-95-13, University of Namur, Belgium, 1993.
[20] J.-L. Hainaut, J. Henrard, J.-M. Hick, D. Roland, Database design recovery, Lecture Notes in Computer Science, vol. 1080, Springer, Berlin, 1996, pp. 272ff.
[21] J.-L. Hainaut, C. Tonneau, M. Joris, M. Chandelon, Transformation-based database reverse engineering, Lecture Notes in Computer Science, vol. 823, Springer, Berlin, 1994, p. 364.
[22] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, J.-L. Hainaut, Program understanding in database reverse engineering, in: Proc. of DEXA'98, Vienna, Austria, 1998.
[23] J. Holle, Ein Generator für integrierte Werkzeuge am Beispiel der objekt-relationalen Datenbankschemamigration, Master's Thesis, Universität-GH Paderborn, Fachbereich 17, Paderborn, Germany, 1997.
[24] F. Hüsemann, Eine erweiterte Schemaabbildungskomponente für Datenbank—Gateways, in: 10. Workshop " 'Grundlagen von Datenbanken", Konstanz, June 1998. Konstanzer Schriften in Mathematik und Informatik Nr. 63, Universität Konstanz, pp. 52–56.
[25] J.H. Jahnke, Management of Uncertainty and Inconsistency in Database Reengineering Processes, Ph.D. Thesis, University of Paderborn, Department of Mathematics and Computer Science, 33095 Paderborn, Germany, September 1999.
[26] J.H. Jahnke, W.Schäfer, A. Zündorf, A design environment for migrating relational to object oriented database systems, in: Proc. of the 1996 Intl. Conf. on Software Maintenance (ICSM'96), IEEE Computer Society, Silver Spring, MD, 1996.
[27] J.H. Jahnke, W. Schäfer, A. Zündorf, Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications, in: Proc. of European Software Engineering Conference (ESEC/FSE), Lecture Notes in Computer Science, vol. B02, Springer, Berlin, September 1997.

[28] J.H. Jahnke, J.P. Wadsack, The varlet analyst: employing imperfect knowledge in database reverse engineering tools, in: Proc. of 3rd Internat. Workshop on Intelligent Software Engineering (WISE-3), Limerick, Ireland, 2000.
[29] J.H. Jahnke, A. Walenstein, Reverse engineering tools as media for imperfect knowledge, in: Proc. of the 7th Working Conference on Reverse Engineering (WCRE 2000), Brisbane, Australia, 2000.
[30] J.H. Jahnke, A. Zündorf, Applying graph transformations to database re-engineering, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation—Application, vol. 2, World Scientific, Singapore, 1999.
[31] P. Johannesson, K. Kalman, A method for translating relational schemas into conceptual schemas, in: F.H. Lochovsky (Ed.), Entity-Relationship Approach to Database Design and Querying, North-Holland, Amsterdam, 1990.
[32] U.A. Johnen, M.A. Jeusfeld, An executable meta model for re-engineering of database schemas, in: P. Loucopoulos (Ed.), Proc. ER'94, Manchester, UK, Lecture Notes in Computer Science, vol. 881, Springer, Berlin, 1994, pp. 533–547.
[33] M. Lefering, A. Schürr, Building tightly integrated software development environments, in: Specification of Integration Tools, Lecture Notes in Computer Science, vol. 1170, Springer, Berlin, 1996.
[34] P. Martin, J.R. Cordy, R. Abu-Hamdeh, Information capacity preserving of relational schemas using structural transformation, Technical Report ISSN 0836-0227-95-392, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, November 1995.
[35] R.W. Mathews, W.C. McGee, Data modeling for software development, IBM Systems J. 29 (2) (1990) 228–235.
[36] S.B. Navathe, A.M. Awong, Abstracting relational and hierarchical data with a semantic data model, in: Proc. of the 6th Intl. Conf. of the Entity Relationship Approach, New York, North-Holland, Amsterdam, November 1987, pp. 305–333.
[37] J.-M. Petit, J. Kouloumdjian, J.-F. Boulicaut, F. Toumani, Using queries to improve database reverse engineering, in: Proc. of 13th Intl. Conf. of ERA, Manchester, Springer, Berlin, 1994, pp. 369–386.
[38] W.J. Premerlani, M.R. Blaha, An approach for reverse engineering of relational databases, Commun. ACM 37 (5) (1994) 42–49.
[39] S. Ramanathan, J. Hodges, Extraction of object-oriented structures from existing relational databases, ACM SIGMOD Record 26 (1) (1997) 59–64.
[40] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, Singapore, 1997.
[41] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object–Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ 07632, 1991.
[42] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, 1st ed., Addison-Wesley, Reading, MA USA, 1999.
[43] B. Schiefer, Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen. PhD Thesis, Universität Karlsruhe, Fakultät für Informatik, FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10–14, D-76131 Karlsruhe, Germany, December 1993.
[44] A. Schürr, A.J. Winter, A. Zündorf, Graph Grammar Engineering with PROGRES, in: W. Schäfer (Ed.), Software Engineering—ESEC'95, Springer, Berlin, 1995.
[45] O. Signore, M. Loffredo, M. Gregori, M. Cima, Reconstruction of er schema from database applications: a cognitive approach, in: Proc. of 13th Intl. Conf. of ERA, Manchester, Springer, Berlin, 1994, pp. 387–402.
[46] G. Snelting, F. Tip, Reengineering class hierarchies using concept analysis, ACM SIGSOFT Software Engineering Notes 23 (6) (1998) 99–110.
[47] P. Sousa, L. Pedro de Jesus, G. Pereira, F. Brito e Abreu. Clustering relations into abstract er schemas for database reverse engineering, in: Proc. of the 3rd European Conference on Software Maintenance and Reengineering (CSMR'99), Amsterdam, NL, IEEE CS, Silver Spring, MD, March 1999, pp. 169–176.
[48] F.N. Springsteel, C. Kou, Reverse data engineering of E-R designed relational schemas, in: Proc. of Databases, Parallel Architectures and their Applications, Springer, Berlin, March 1990, pp. 438–440.
[49] The ProgresDeveloper Team, *The Progres Language Manual Version 9.2*, Lehrstuhl fuer Informatik III, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany, 1999.

[50] M. Tresch, Evolution in Objekt-Datenbanken, Teubner, Stuttgart, 1995.
[51] A. Umar, Application (Re)Engineering—Building Web-Based Applications and Dealing with Legacies, Prentice-Hall International, London, UK, 1997.
[52] S. Vinoski, Corba: integrating diverse applications within distributed heterogeneous environments, IEEE Commun. Magazine 14 (2) (1997) 46–55.
[53] J.P. Wadsack, Inkrementelle Konsistenzerhaltung in der transformationsbasierten Datenbankmigration, Master's Thesis, University of Paderborn, Department of Mathematics and Computer Science, 33095 Paderborn, Germany, 1998.