Integration of Analysis and Redesign Activities in Information System Reengineering

Jens H. Jahnke, Jörg Wadsack
AG-Softwaretechnik, Fachbereich 17, Universität Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany;
e-mail: [jahnke|maroc]@uni-paderborn.de

Abstract

Emerging key technologies like the World Wide Web, object-orientation, and distributed computing enable new applications, e.g., in the area of electronic commerce, management 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. However, a major limitation of current approaches is that they impose a strictly phase-oriented, waterfall-type reengineering process, without the 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 the migration process. In this paper, we present an approach to incremental consistency management that allows to overcome this limitation by integrating reverse and forward engineering activities in an intertwined process. The described mechanism is based on a formalization of redesign transformations by graph rewriting rules and has been implemented in the Varlet reengineering environment.

1. Introduction and related work

An 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. Emerging key technologies like the *World Wide Web* (Web), *Object-Orientation* (OO), *Client/Server* (CS) *applications*, and *open system standards* (e.g., CORBA [23], DCE [5]) greatly influence modern business processes. Besides 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 [22]. While new company start-ups are able to purchase information systems (IS) that take advantage of the latest technology, longer established enterprises have to modify pre-existing IS to fit new requirements and exploit emerging technologies. This is often a challenging task because many IS have evolved over several generations of programmers and comprise only obsolete or no documentation. Such applications are usually called *legacy information systems* [22].

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], most approaches focus on database *schema analysis* [8,20,16,9] (reverse engineering) and/or *schema translation and redesign* (forward engineering) [3,4,17,10,15,6,14].

One of the most important limitations of current CARE tools is that they do not consider the evolutionary and exploratory nature of the reengineering process [7]. 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 analysis and redesign steps occur frequently: When a reengineer learns more about the abstract design of a legacy system, (s)he often refutes some initial assumptions or does some 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 IS during this period. These modifications have to be reflected in the migrated target system, which demands for iterations in the reengineering process. Using current tools, the reengineer loses the work (s)he has done in forward-engineering when changing the legacy IS, in order to re-establish consistency. In this paper, we describe an integrated CARE environment that overcomes the described problem. Our approach is

Published in *Proc. of the 3rd European Conference on Software Maintenance and Reengineering (CSMR'99), Amsterdam, NL.*

based on a formalization of redesign operations by *graph rewriting rules* [18]. 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 defines a formal basis of our approach and describes the main implementation concepts of our CARE environment. Finally, Section 4 closes with some concluding remarks.

2. A reengineering case study

The following case study reflects some of our experiences with industrial projects. 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. Now, the IT department plans to employ Internet-technology to establish a distributed Web-based *marketing information system* (MIS) 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 order to implement the desired MIS the legacy relational database schema has to be well-understood. Unfortunately, the legacy IS is hardly documented and the programmers have left the company. Thus, the PDIS has to be *analyzed* to yield a semantically enriched source schema, which can be *translated* (and *redesigned*) into a conceptual target schema that is suitable for the new MIS. In general, this is an evolutionary and explorative process, as indicators for abstract design concepts are often hidden in different parts of the legacy IS, including its data, code, physical

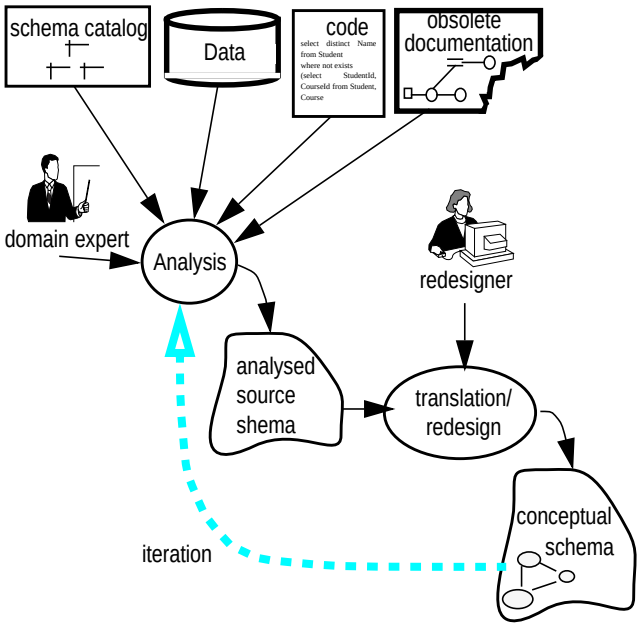

Figure 1. Schema reengineering process

schema, and its (obsolete) documentation (cf. Figure 1). Furthermore, many legacy IS comprise arcane coding concepts (e.g., variant records) and various kinds of optimization structures.

The following description demonstrates how this evolutionary process is supported by our CARE environment *Varlet* [12], i.e., how *Varlet* preserves the consistency between the analyzed source schema and the redesigned conceptual target model.

Step 1: schema analysis, translation, and redesign

Figure 2 shows a screenshot of the *Varlet analyzer view* that displays a detail of the legacy database schema. At this, each box represents a relational table, while foreign keys are

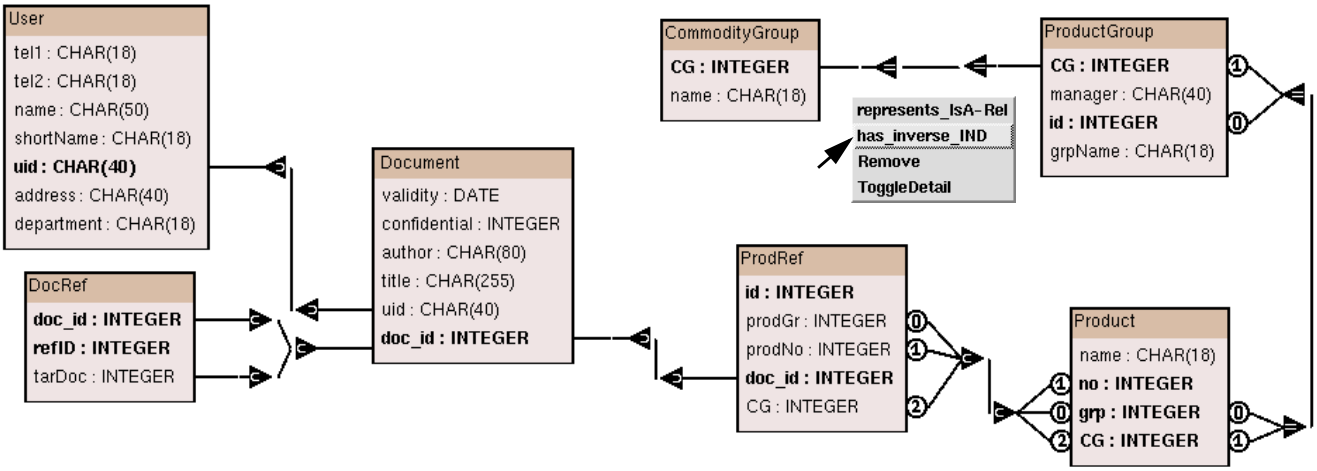

Figure 2. Case study: analysed relational legacy schema (detail)

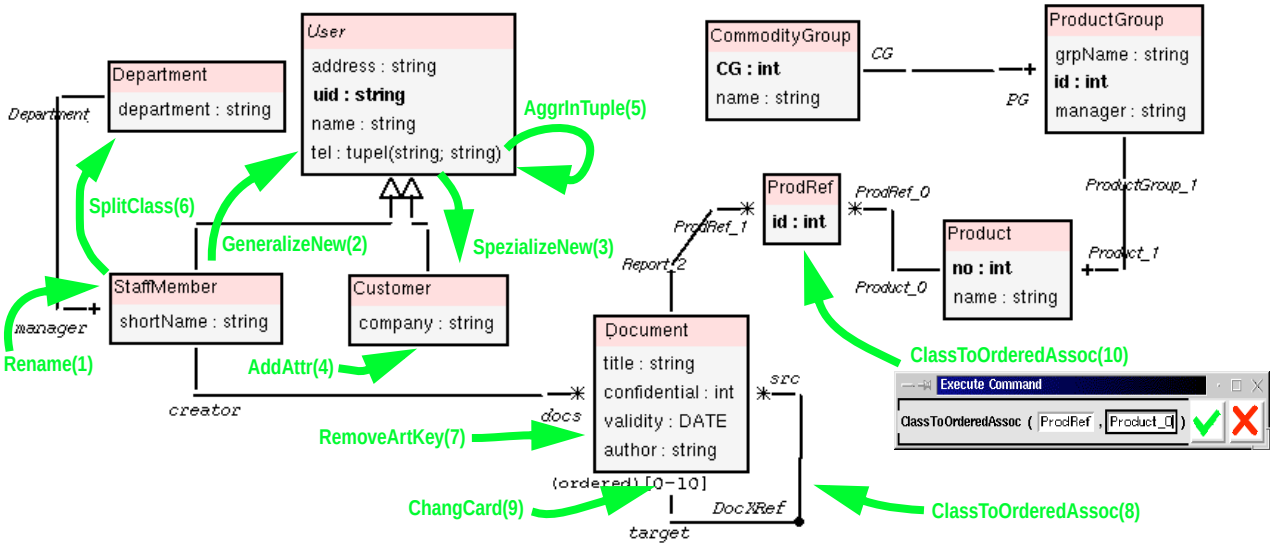

Figure 3. Redesigned conceptual model

represented as directed lines 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 legacy IS. In case of our case study, let us assume the reengineer has already recovered the constraints shown in Figure 2[1]. 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 [6] 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*.

After the PDIS has been analyzed, *Varlet* automatically translates the semantically enriched schema into an object-oriented conceptual model. This initial translation can be redesigned and extended by using a set of predefined *redesign operations*. Figure 3 shows a sample resulting object model for our scenario. At this, bold grey arrows denote ten sample applications of redesign operations the reengineer has performed after the initial conceptual translation. For example, the initially created class *User* has been renamed (1) to *StaffMember* and has been generalized (2) by a new 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 Figure 3 exemplifies that redesign operations are interactively invoked by actualizing their formal parameter lists.

1. See [13] and [11] for more information about the actual analysis process in *Varlet*.

Step 2: schema completion and re-translation

Let us further assume, that by investigating 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 different variants of rows in this table (cf. Figure 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 variant 4) in table *ProdRef* with equal key values. In

variant	id	prodNo	prodGr	doc_id	CG
1	...	...	...	...	...
2	...	null	...	...	...
3	...	null	null	...	...
4	...	null	null	...	null

Figure 4. Variants of table *ProdRef*

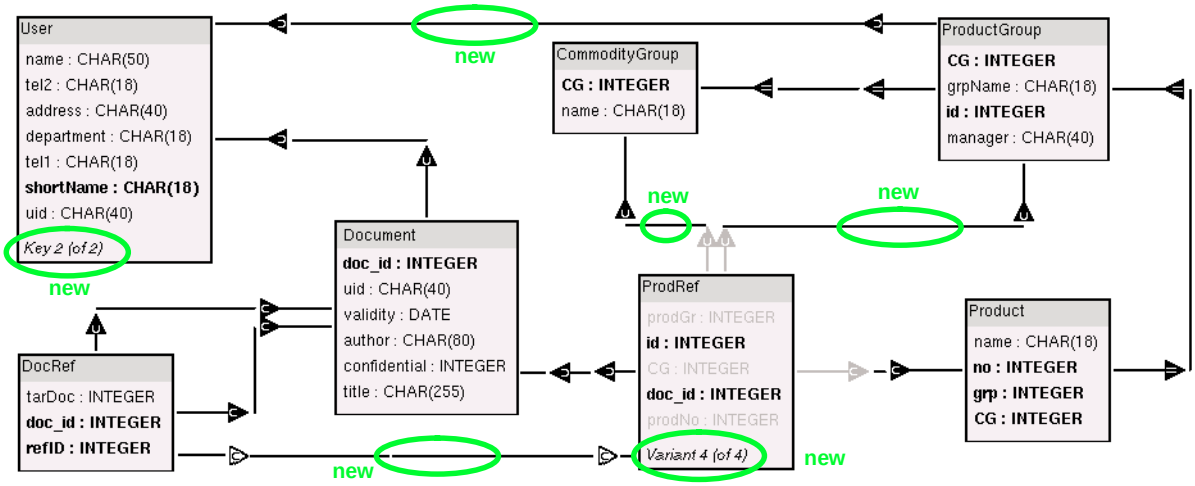

Figure 5. Completed relational schema

addition, the reengineer finds out that *shortName* is an alternative key of table *User*, borrowed by table *ProductGroup* in column *manager*.

The completed relational schema is given in Figure 5. In this figure, we used ovals to mark the differences between the completed relational schema and the first analysis result (cf. Figure 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. 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 Figure 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 created conceptual model has been lost. Using currently existing, waterfall-oriented tools, the reengineer has two options to try to re-establish 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 determine the impact of the modifications on the redesigned conceptual model manually. Both alternatives are unsatisfactory for larger schemas: the first solution will most likely force the reengineer to manually redo redesign operations, (s)he has already performed once, while the second solution is error prone.

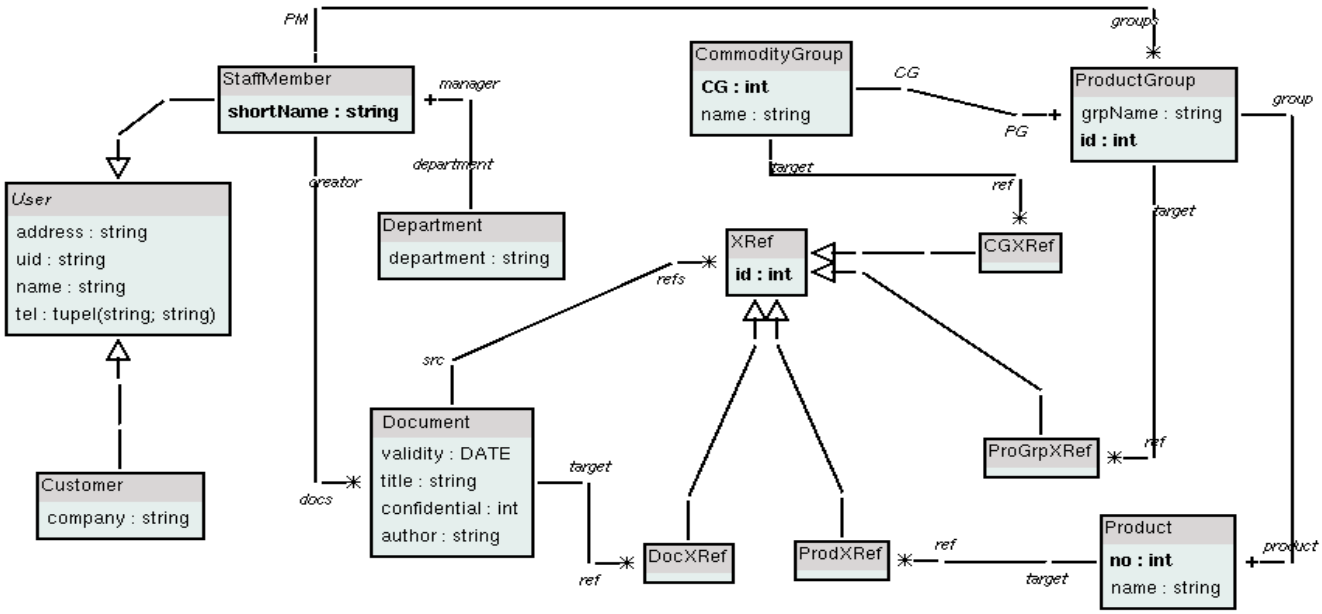

Figure 6. Updated conceptual model

Our approach overcomes this problem, by providing an incremental mechanism to re-establish consistency between the legacy schema and the redesigned conceptual model. Figure 6 shows the new conceptual model, which has been updated automatically according to the new information about the legacy schema. The environment determined that operations 8, 9 and 10 have to be undone. 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 precondition of operation *ClassToOrderedAssoc*. Operation 9 is no longer applicable, as it depends on the applicability of operation 8.

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 database reengineering environment. In the next section, we will describe implementation concepts for such a component. The main idea is to specify redesign operations as graph rewriting rules 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.

3. Implementation Concepts

3.1 The migration schema

In our reengineering environment we employ *attributed abstract syntax graphs* (ASG) to represent the legacy schema and the corresponding conceptual model. Both ASGs are connected via an additional mapping structure, which reflects their interdependencies. This entire graph structure is called the *migration schema*. Figure 7 shows a simplified detail of the migration schema for our sample scenario. The ASG of the relational schema is given on the left side of Figure 7, while its right side shows the representation of the conceptual model. The syntactic root of the relational schema is represented by a node of type *SQLSchema* carrying the name *"PDIS"* of the example schema as an attribute. Starting with this root node, the syntactic structure of the legacy schema is represented by all nodes that are transitively reachable by traversing edges of type *c* (contains). A table definition is represented by a node of type *table* with outgoing *c* edges leading to the contained attribute and foreign key definitions. The ASG of the conceptual model has a similar structure. The mapping structure represents the correspondences between relational and conceptual schema elements, e.g., a node of type *MapTable* shows that table *Document* has been mapped to an equally named class. The same applies for table *User*, but for

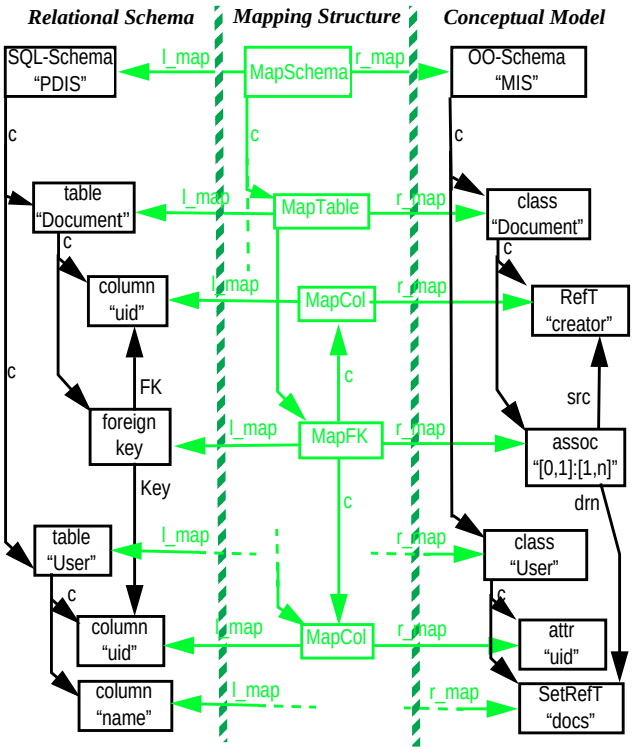

Figure 7: Internal graph representation of the *migration schema*

simplicity we left out the second *MapTable* node. The foreign key between both tables has been mapped to a 1:n-association between the corresponding classes (cf. node *MapFK*). Note, that no redesign operation has been performed so far, i.e., Figure 7 shows the situation after the initial translation of the relational schema into the conceptual model.

3.2 Schema translation and redesign by graph rewriting

In our approach, the initial translation of the legacy schema and all conceptual redesign operations are formally specified by *graph rewriting rules* [18]. In order to introduce them, we need the formal concept of a graph[1].

Definition 1. Graph

*$G := (N, E, t_N, t_E, s, t)$ is a **graph** over two given label sets L_N, L_E with:*

- *$N(G):=N$ and $E(G):=E$ are finite sets of nodes and edges,*
- *$t_N(G):N \rightarrow L_N$ and $t_E(G):E \rightarrow L_E$ are their typing functions,*
- *$s(G):E \rightarrow N$ and $t(G):E \rightarrow N$ assign each edge its source and target.*

1. For simplicity, our formal definition of a graph (and a graph rewriting rule) does not consider attributes of graph objects.

Definition 2. Graph rewriting rule

*A **graph rewriting rule** o:= (P, Q) is a tuple of graphs (P and Q) over the same sets of node and edge labels. The graphs P(o):=P and Q(o):=Q are called the left- and right-hand side of o and represent the pre- and postcondition of the rule, respectively. P(o) describes which elements (nodes and edges) a graph G must contain such that o can be applied. If G contains a subgraph S that is a valid match for P(o), Q(o) describes how S shall look like after the application. P(o) and Q(o) can have common objects. These objects are preserved during the application of o. Objects in P(o) that do not occur in Q(o) are deleted, while objects in Q(o) which do not occur in P(o) are added.*

The following two definitions provide a formal criterion for the applicability of graph rewriting rules. We employ this criterion to (re-)validate the preconditions of redesign operations when a modification of the legacy schema has to be propagated.

Definition 3: Match of a graph rewriting rule

*Given a graph G and a graph rewriting rule o: (P, Q), a morphism in:P→G identifies a **match** of o in G, iff:*

- *objects that will be deleted can uniquely be identified, i.e.* $\forall x \in P \backslash Q, x' \in P: in(x)=in(x') \Rightarrow x = x'$
- *there are no dangling edges after the deletion, i.e.* $\forall n \in N(P) \backslash N(Q), \forall e \in E(G)$: $(s(e)=in(n) \vee t(e)=in(n)) \Rightarrow \exists e' \in E(P) \backslash E(Q) : in(e')=e$

Definition 4: Application of a graph rewriting rule

*An **application** of a graph rewriting rule o: (P, Q) to a graph G is represented by a tuple* $\overline{o}$*: (in, out) with two morphisms in:P→G and out:Q→ G' , where*

- *in is a match for o in G,*
- *out is a match for o':(Q,P) in G',*
- $\forall x \in P \cap Q$*: in(x)=out(x), and*
- *G \ (in(P\Q)) = G' \ (out(Q\P)).*

The application of a graph rewriting rule o on a graph G with result G' will be denoted as $G'=G^{\downarrow \overline{o}}$.

A sample graph rewriting rule that is used during the initial translation of the relational schema into the conceptual model is given in Figure 8. It describes that a relational table can initially be mapped to a class. The element on the left-hand side are identically preserved at the right-hand side. In addition, the right-hand side contains two new nodes (8 and 9) and four new edges to represent the new class and the migration schema. The transfer clause at the bottom of Figure 8 passes the name of the mapped table to the newly created class. The information maintained in the

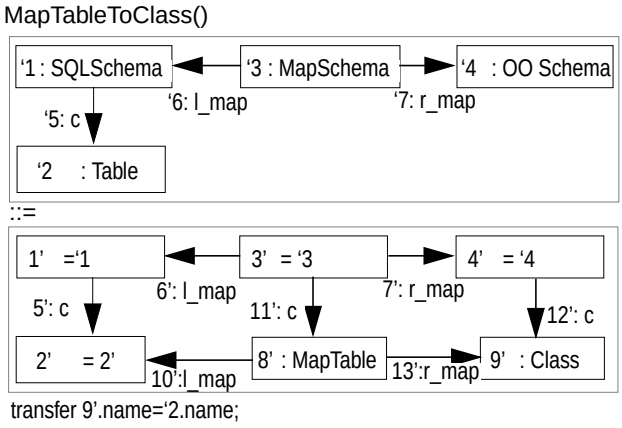

Figure 8. Graph rewriting rule *MapTableToClass*

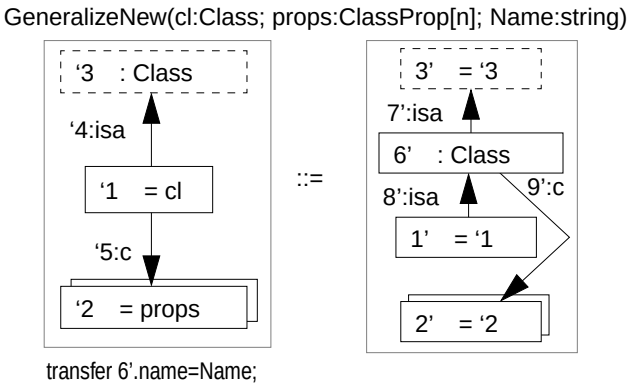

Figure 9. Graph rewriting rule *GeneralizeNew*

mapping structure is used during the initial translation to apply the mapping rules in the right order. A discussion of this algorithm is out of the scope of this paper and can be found in [19].

Figure 9 shows another graph rewriting rule that specifies one of the redesign operations (*GeneralizeNew*), we used in our case study. Dashed boxes represent optional nodes, while node sets are represented by stacked boxes. The left-hand side of *GeneralizeNew* looks for a node of type *Class* (1) that aggregates a set of nodes of type *ClassProp* (2) via edges of type *c*. Furthermore, node 1 might have an outgoing edge labelled *isa* (4) to another node of type *Class* (3). In contrast to rules used for the initial schema translation, redesign rules are applied manually. At this, the reengineer chooses the application context of a redesign rule by actualizing its formal parameters (*cl*, *props*) by references to selected schema components. An application of *GeneralizeNew* creates a node (6) that represents a new superclass of the selected class. Moreover, the two edges on the left-hand side are replaced by new edges representing the migration of the selected class properties (2) to the generalization.

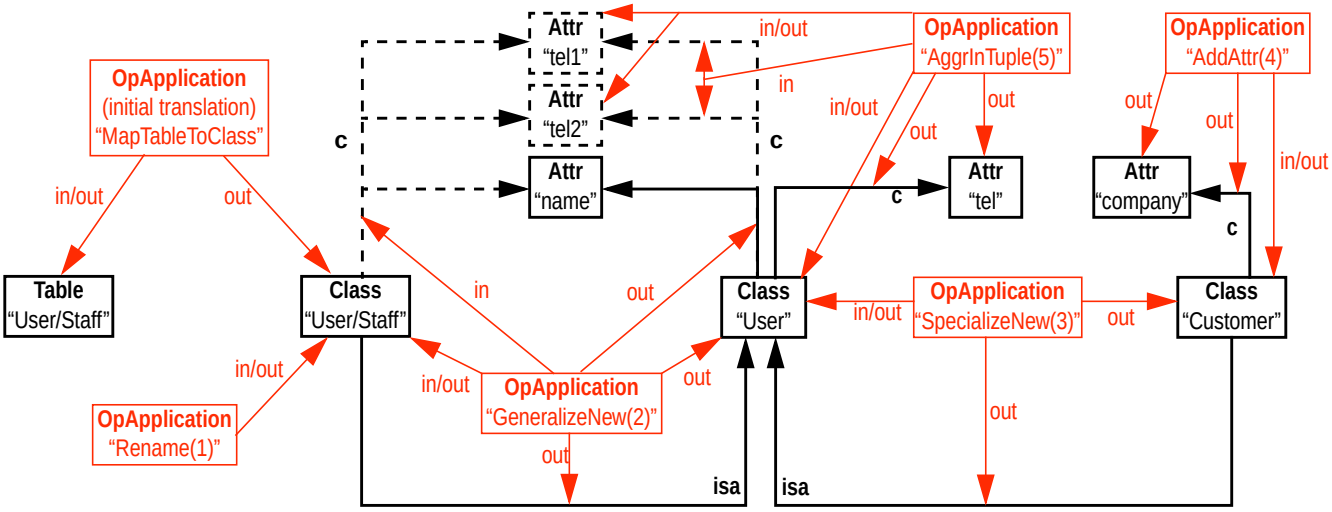

Figure 10. History graph (detail)

The purpose of our consistency management component is to propagate modifications of the legacy schema to the entire migration schema. This can be achieved by re-evaluating the pre-conditions of all applied (redesign) operations that are affected by the modification. The case study shows that often redesign operations depend on the result of previously performed operations. By using their representation as graph rewriting rules, we are able to define a dependency relation that will be used to determine the set of affected operation applications:

Definition 5. Structural dependency

Let Ω *denote a set of applications of graph rewriting rules, with* $\overline{o}$*:(in,out),* $\overline{o}$*':(in',out')*$\in\Omega$*. Application* $\overline{o}$*'* ***structurally depends*** *on* $\overline{o}$ *in* Ω *(denoted as* $\overline{o}' \dashv^{\Omega} \overline{o}$*),*
iff: $\overline{o}=\overline{o}' \vee \exists\, x\in P(o'), y\in Q(o)\backslash P(o)$:

$$in'(x)=out(y) \vee \exists \overline{o}''\in\Omega : \overline{o}' \dashv^{\Omega} \overline{o}'' \wedge \overline{o}'' \dashv^{\Omega} \overline{o}$$

To propagate modifications, the *Varlet* environment maintains a graph that not only represents the current state of the migration schema, but also provides information about all applied operations and their structural dependencies. This graph is called the *history graph*, because it reflects the editing history. Figure 10 shows a detail of a history graph, that represents an application of mapping rule *MapTableToClass* and the first five applications of redesign operations in our case study (cf. Figure 3). At this, objects of the migration schema have black color, while grey color is used to represent operation applications. Objects of the migration schema that have actually been consumed by redesign operations (i.e. that have an ingoing *in* edge) remain in the history graph. They are marked using dashed lines.

Definition 6. History graph

A ***history graph*** *is a graph that includes the current migration schema as a subgraph. Moreover, it contains nodes and edges that represent all applications of (mapping and redesign) operations. Given a history graph H:=(N, E,* t_N*,* t_E*, s, t) the current migration schema MS(H):=(N', E',* t_N*,* t_E*, s, t) is defined as:*

- $N' := \{n\in N \mid (\exists e\in E: t(e)=n \wedge t_E(e)='out') \wedge$
  $(\forall n'\in N, e\in E: t(e)=n \wedge s(e)=n' \wedge t_E(e)='in'$
  $\Rightarrow \exists e' \in E: t(e')=n \wedge s(e')=n' \wedge t_E(e')='out')\}$
- $E' := \{e\in E \mid s(e), t(e) \in N'\}$

Now we can formulate an algorithm that uses a history graph to propagate a modification of the legacy schema to the entire migration schema. Again, the modification of the legacy schema is formally represented by an application of a graph rewriting rule. The algorithm in Figure 11 takes the current history graph and a given modification ($\overline{m}$) as inputs and returns the updated history graph. At first, it uses the structural dependency relation to determine all applied operations that are affected by $\overline{m}$. Then it uses the *time* attribute, which reflects the chronological order of applied operations, to select the oldest operation application ($\overline{o}$). Subsequently, it re-evaluates if $\overline{o}$ is still a valid application after the modification. If this is not the case, it determines all applied operations that structurally depend on $\overline{o}$ and removes them from the history graph. Finally, $\overline{o}$ is removed from the set of operation applications to be validated (Ω^a). This is iteratively done until this set is empty.

While propagating a modification *Varlet* logs all redesign operations that are no longer valid. The purpose of this log is to explicitly inform the reengineer about changes that

have been performed to the conceptual model in order to re-establish consistency. If operations applied by the initial schema translation have been undone a number of unmapped legacy schema elements result. In this case, the initial translation algorithm is called after the propagation to find a new translation of these elements.

Propagate (H, $\overline{m}$:(in_m,out_m)) returns H'
H and H' are the history graph before respectively after a modification $\overline{m}$.
$\overline{m}$ is a modification of MS(H) (i.e., modification of the migration schema)
begin
let Ω denote the set of all operation applications represented in H;
Ω^a:={$\overline{o}$:(in,out)$\in \Omega$ | x$\in$ P(m), y$\in$ P(o): in(y)=in_m(x)};
// get all directly affected operation applications
while $\Omega^a \neq \emptyset$ **do**
let $\overline{o}$:(in,out)$\in \Omega^a$ with $\overline{o}$.time = Min({$\overline{o}$'.time | $\overline{o}$'$\in \Omega^a$});
// get operation application with min. time stamp
if *in* represents a match of o in (MS(H)$\downarrow^{\overline{m}}$) according to Def. 3 **then**
Ω^d:= {$\overline{o}$'$\in \Omega$ | $\overline{o}$' $\downarrow^{\Omega}$ $\overline{o}$ }
remove all $\overline{o}$'$\in \Omega^d$ from H
fi
remove $\overline{o}$ from Ω^a
loop
return H as H'
end

Figure 11. Change propagation algorithm

4. Conclusions

Providing tools that allow for an evolutionary 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 IS. A formal specification of all operations applied to a legacy schema allows to propagate modifications in case of cycles between analysis and redesign activities. In general, many transformation-based techniques are suitable to implement our approach [2]. We have chosen graph rewriting rules, because they are executable *and* intuitively well-understandable.

We employ the *Progres* [21] (programmed graph rewriting systems) environment to implement the current version of the *Varlet* consistency management component. With *Progres* executable code can be automatically generated from specified graph rewriting rules. Thus, the *Varlet* environment facilitates the customization of (redesign) operations by simply adding new or changing existing graph rewriting rules. The current (graphical) *Progres* specification consists of 300 pages. Form 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.

We have evaluated *Varlet* in an industrial project that is similar to the sample scenario described in this paper. The real case study deals with 79 relational tables. Our approach scales well, because the reengineer interactively determines the application context for redesign operations (cf. Figure 3). Consequently, matches for applied graph rewriting rules can be identified in constant time (if applicable). The same applies for the actual applications of graph rewriting rules and the corresponding update operations on the history graph. In case of an iteration between analysis and redesign activities, the change propagation algorithm presented in Figure 11 obviously has the complexity *O(n)*, where *n* denotes the number of redesign operations performed to the initial schema translation. Our current research focus is on generalizing the presented approach to also consider the dynamic part of a legacy IS, i.e., its application code.

Acknowledgments

We would like to thank Barbara Bewermeyer, Ulrich Nickel, Wilhelm Schäfer, Felix Wolf, and Albert Zündorf for many fruitful discussion and valuable comments. Furthermore, we would like to thank Tarja Systä for her great support in editing this paper.

References

[1] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[2] I. Baxter. Tutorial on Transformation System - Transformation Technology Bibliography. *Fifth International Conference on Software Reusability (ICSR5*), *Victoria, B.C., Canada*, June 1998.

[3] Andreas Behm, Andreas Geppert, and Klaus R. Dittrich. On the migration of relational schemas and data to object-oriented database systems. In *Proc. 5th International Conference on Re-Technologies for Information Systems, Klagenfurt, Austria*, December 1997.

[4] J. Fong. Converting relational to object-oriented databases. *ACM SIGMOD Record*, 26(1), March 1997.

[5] Open Software Foundation. *Introduction to OSF/DCE*. Prentice Hall, New Jersey, 1992.

[6] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93. In *Proc. of the 4th Int. Conf. of on Deductive and Object-Oriented Databases 1995*, 1995.

[7] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, and D. Roland. Requirements for information system reverse engineering support. Technical Report RP-95-13, University of Namur, Belgium, 1993.

[8] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, and J.-L. Hainaut. Program understanding in database reverse engineering. Technical Report RP-98-004, Institute d'Informatique, University of Namur, Belgium, 1998.

[9] Himel Inc, 17153 President Drive, Castro valley, CA 94546,

USA. *DBInformer User's Manual*, 1997.

[10] F. Hüsemann. Eine erweiterte Schemaabbildungskomponente für Datenbank–Gateways. In *10. Workshop "'Grundlagen von Datenbanken"'*, pages 52–56, Konstanz, June 1998. Konstanzer Schriften in Mathematik und Informatik Nr. 63, Universität Konstanz.

[11] J. H. Jahnke and M. Heitbreder. Design recovery of legacy database applications based on possibilistic reasoning. In *Proceedings of 7th IEEE Int. Conf. of Fuzzy Systems (FUZZ'98). Anchorage, USA.*. IEEE Computer Society, May 1998.

[12] J. H. Jahnke, W. Schäfer, and A. Zündorf. A design environment for migrating relational to object oriented database systems. In *Proc. of the 1996 Int. Conference on Software Maintenance (ICSM'96)*. IEEE Computer Society, 1996.

[13] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer, September 1997.

[14] P. Martin, J. R. Cordy, and R. Abu-Hamdeh. Information capacity preserving of relational schemas using structural transformation. Technical Report ISSN 0836-0227-95-392, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, November 1995.

[15] ONTOS Inc., Three Burlington Woods, Burlington, MA, USA. *ONTOS Object Integration Server for Relational Databases 2.0 - Schema Mapper User's Guide*, 2.0 edition, 1996.

[16] W. J. Premerlani and M. R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, May 1994.

[17] S. Ramanathan and J. Hodges. Extraction of object-oriented structures from existing relational databases. *ACM SIGMOD Record*, 26(1), March 1997.

[18] Grzegorz Rosenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[19] Andreas Schuerr. Specification of graph translators with triple graph grammars. Technical report, RWTH Aachen, 1994.

[20] O. Signore, M. Loffredo, M. Gregori, and M. Cima. Reconstruction of er schema from database applications: a cognitive approach. In *Proc. of 13th Int. Conference of ERA, Manchester*, pages 387–402. Springer, 1994.

[21] A. Schürr and A. J. Winter and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer (ed.) *Software Engineering - ESEC '95*. Springer Verlag, 1995

[22] Amjad Umar. *Application (Re)Engineering - Building Web-Based Applications and Dealing with Legacies*. Prentice-Hall International, London, UK, 1997.

[23] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.