The *Varlet Analyst*: Employing Imperfect Knowledge in Database Reverse Engineering Tools

Jens H. Jahnke
University of Victoria
Department of Computer Science
P.O. Box 3055
V8W3P6 Victoria, B.C., Canada
jens@acm.org

Jörg Wadsack
University of Paderborn
Department of Mathematics and Computer Science
Warburger Str. 100
33098 Paderborn, Germany
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 existing database (DB) applications to take advantage of these technologies. Various computer-aided reengineering tools have been developed to reduce the complexity of the reengineering task. However, most of these approaches presume complete structural and semantical information about the DB schema and provide only little support for earlier analysis activities that aim to obtain this information. Such activities are mainly performed manually with the aid of very simple, loosely-coupled tools for textual search or data analysis. The reengineer has to judge and combine many different semantic indicators from various sources of information to recover a complete DB schema. In this paper, we present a flexible tool that aims to support the reengineer in these reverse engineering activities. Unlike other tools, our approach does not force the reengineer to follow a strict process or to enter only consistent information. On the contrary, our tool adopts the mental model of its user and deals with imperfect information (uncertainty and contradiction) explicitly.

1. Introduction

Today's information technology (IT) undergoes dramatic *mass changes* [1] due to urgent requirements like the coming of the next millennium (*Year-2000-problem*) [2], the European currency union [3], and emerging technologies like the *World Wide Web. Electronic Commerce* is about to become one of the key business technologies for the next decade. While new company start-ups are able to purchase modern database (DB) technology to develop information systems that meet these new requirements, longer established enterprises have to deal with pre-existing DB applications. In many cases, such *legacy* DB applications have evolved over several generations of programmers and lack a sufficient technical documentation. Still, they maintain a vast amount of valuable business data and their functionality is often critical for the mission of the enterprise. Consequently, a complete replacement of these systems is virtually impossible or at least implies a significant risk.

In order to solve this problem, during the recent decade there has been increasing effort to develop methods and tools to *reverse engineer* legacy DBs. The general goal of DB reverse engineering (DBRE) activities is to recover a conceptual design for an implemented DB schema. Such a conceptual design provides a high level of abstraction that is prerequisite to achieve a variety of assessment and maintenance activities. The DBRE process mainly consists of two subsequent phases, namely *schema analysis* and conceptual *schema translation* (cf. Figure 1). In the schema

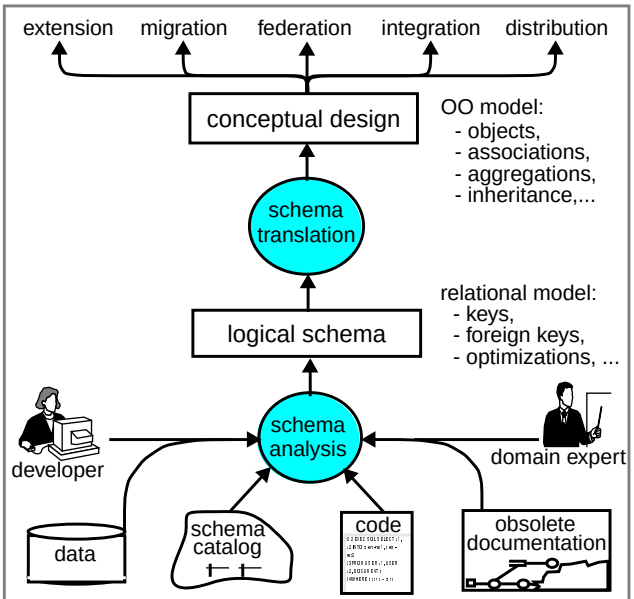

Figure 1. Database reverse engineering process

analysis phase, the reengineer aims to reconstruct a logical schema that is structurally complete and semantically enriched [4]. In case of a relational DB the reengineer aims to identify and classify key and foreign key dependencies, not-null constraints, optimization structures, and denormalizations. Typically, some important structural and semantical information is not represented explicitly in the schema catalog of many legacy DBs. However, *implicit* indicators for such information might be found in different parts of the legacy application, including its schema catalog, data, procedural code, and (obsolete) documentation. The reengineer has to find and combine these indicators to yield the desired logical schema. For larger systems this is a complex and laborious task that requires a lot of experience. Once, the logical schema of a legacy DB has been recovered it can be translated into a conceptual data model, e.g., an object-oriented (OO) or extended entity-relationship (EER) model [5].

Various *computer aided reverse engineering* (CARE) tools have been developed in industry and academy to support the DBRE process, e.g., [6, 7,8, 9,10,4,11,12]. Still, most of these approaches have their primary focus on the second phase (conceptual schema translation), i.e., they presume the existence of a complete logical schema. Only a few tool-based approaches provide (limited) support for the activity of legacy schema analysis, e.g., [10, 12]. The reason for this unbalanced situation is that conceptual schema translation deals with canonical operations that can be formalized based on the well-explored theories, e.g., transformation systems [13, 6, 10, 11]. On the opposite, legacy schema analysis is a cognitive activity. CARE tools that are of practical use in this activity have to offer solutions for two inherent problems, namely *imperfect knowledge* and *variety of application contexts*.

The first problem addresses the fact that schema analysis employs various heuristics and vague concepts that deliver uncertain and partial contradicting analysis results. Moreover, humans (e.g., reengineers, developers, domain experts) have uncertain assumptions about the internal realization of legacy systems. A suitable CARE tool has to represent, propagate, and indicate such imperfect knowledge and guide the user to a complete and consistent result.

The second problem (variety of application contexts) reflects the fact that legacy DB applications comprise idiosyncratic coding styles and naming conventions. Furthermore, they are based on diverse hard- and software platforms, data models, etc. Lack of customizability has been recognized as *"the single most common limiting factor in using tools for software analysis and transformation"* [14]. Even though current compiler technology provides mechanisms to generate parsers for different programming languages based on abstract specifications, most existing CARE tools still employ general-purpose programming languages to implement DBRE heuristics and analysis processes. As a consequence, these heuristics and processes can hardly be customized for changing application contexts.

In [15], we have proposed concepts to overcome the two aforementioned limitations of current DBRE tools: we introduced *Generic Fuzzy Reasoning Nets* (GFRN) as an abstract formalism to specify and customize DBRE knowledge and analysis processes. Furthermore, we have employed possibility theory to develop an inference engine that executes GFRN specifications and manages imperfect DBRE knowledge [16]. This paper describes a DBRE tool (*the Varlet Analyst*) that implements these concepts and guides the reengineer in an exploratory and evolutionary schema analysis process.

The rest of this paper is structured as follows. In Section 2, we describe the customizable, semi-automatic schema analysis process that is supported by our tool in more detail. In Section 3, we exemplify the customization and application of our tool with a sample scenario that has been taken from one of our industrial case studies. Section 4 gives some insight in the architecture and internal realization of the *Varlet Analyst*. Finally, in Section 5 we discuss related work and give concluding remarks about the experiences with our approach.

2. Customizable tool support for legacy schema analysis

Our approach to a customizable, semi-automatic schema analysis process is presented as a data flow diagram in Figure 2. Activities that belong to the *customization process* are displayed with a grey background. In this process, the reengineer investigates the legacy DB in order to determine the specific application context of the *Varlet Analyst*. The result of this *domain analysis* activity is a set of technical and non-technical characteristics, e.g., properties of the employed software platform, the size of the legacy system, and applied coding or naming conventions. Based on these characteristics the reengineer *specifies* or adapts domain-specific heuristics and analysis operations which will be applied in the schema analysis process. This knowledge and process is formally represented by a GFRN specification.

After the *Varlet Analyst* has been customized with respect to its current application context it can be employed to support the reengineer in analyzing the schema of the legacy DB. This analysis activity is performed in a semi-automatic process. At first, automatic analysis operations are applied to different legacy software artifacts including the legacy DB's schema catalog, procedural code, and the available data. The

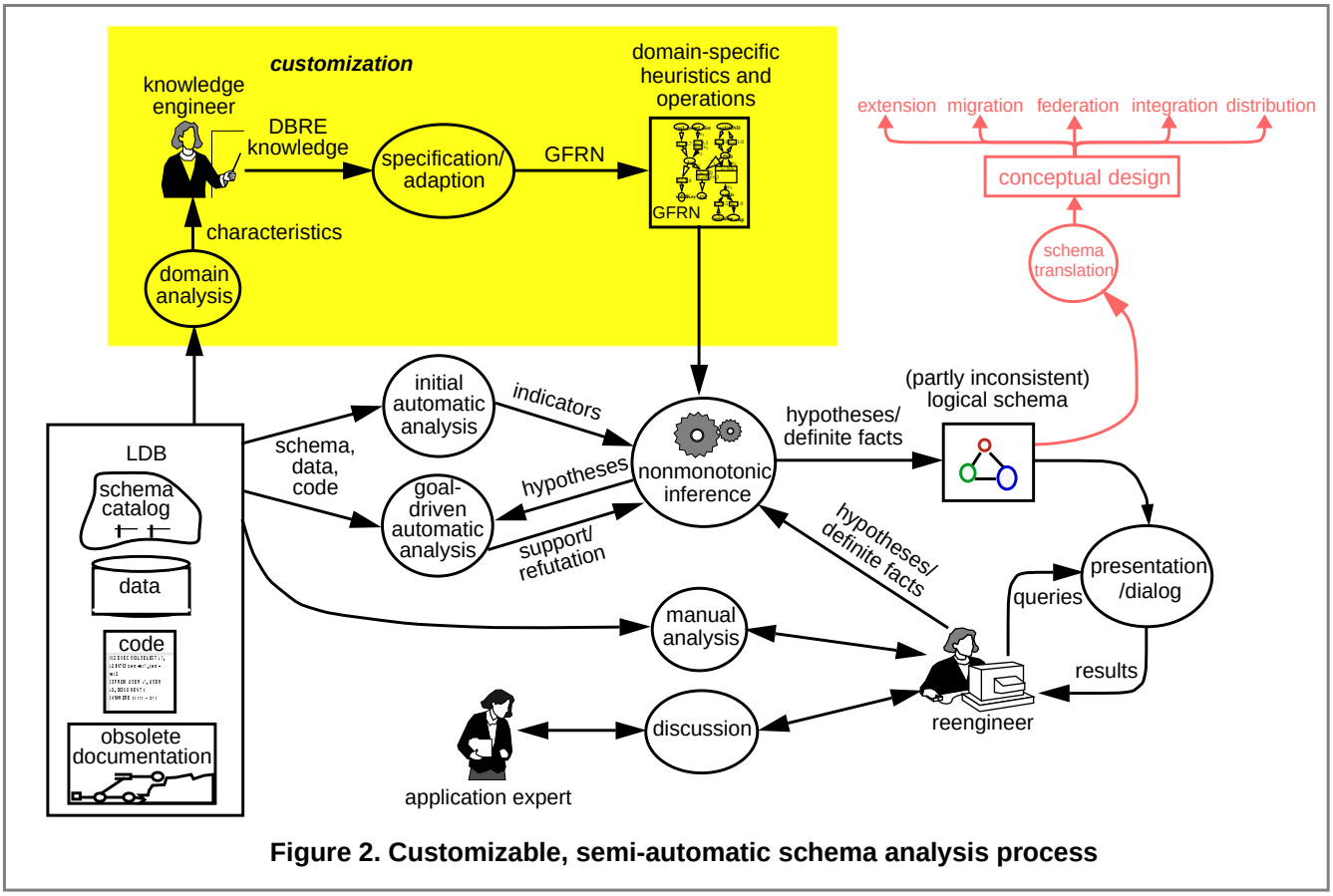

Figure 2. Customizable, semi-automatic schema analysis process

result of this *initial automatic analysis* is a set of (situation-specific) facts about the legacy DB (cf. Figure 2). Subsequently, these facts are taken as indicators which are combined with domain-specific heuristics specified in the GFRN to infer new situation-specific knowledge about the legacy DB. This situation-specific knowledge might comprise definite facts as well as uncertain and inconsistent hypotheses. Again, some of these hypotheses might be refutable using automatic analysis operations. We call such analysis operations *goal-driven* because they are performed "on-demand" to support or refute intermediate assumptions. Again, the GFRN specification determines which goal-driven analysis operations are available and when they are performed.

The output of this automatic inference step is a logical schema which might still partially be inconsistent and incomplete. This schema is presented to the reengineer in a *dialog* process that provides interactive query facilities to indicate the sources of such imperfect knowledge. The reengineer might discuss controversial information with application experts (e.g., developers or operators) and do further manual investigations. As a result of these manual activities the reengineer will enter additional hypotheses or definite facts about the legacy DB. Subsequently, the automatic inference is resumed, i.e., new knowledge is inferred and (goal-driven) analysis operations might be performed to validate hypotheses. The described semi-automatic schema analysis process is iterated until the information about the logical schema is consistent (and complete). In the next section, we give an application-driven illustration of the described process with a reverse engineering example scenario. Detailed information on the the theory behind the described GFRN formalism and the implementation of its inference engine are out of the scope of this paper and have been published in [15, 16, 18].

3. Tool application scenario

In this section, we use a sample scenario to present a tool (the *Varlet Analyst*), that implements the approach described in the previous section. The sample scenario deals with a legacy *product and document information system* (PDIS) of an international enterprise that produces a great variety of drugs and other chemical goods. Traditionally, this system has been used by members of the central hotline at the company headquarter. Recently, the IT department has decided to employ Internet-technology to establish a

distributed Web-based *marketing information system* (MIS) as an extension of 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 develop the data access layer that implements the integration of the existing PDIS with the Web-server the legacy DB schema has to be well understood. Unfortunately, important structural information like referential integrity constraints and alternative key dependencies are not specified explicitly in the physical schema of PDIS. Furthermore, the schema is hardly documented and the responsible developers have left the company. Thus, PDIS has to be *analyzed* to obtain a structurally and semantically complete logical schema that is a prerequisite for the desired integration. In the next two sections, we exemplify the *customization* (Section 3.1) and *application* (Section 3.2) of the *Varlet Analyst* to this schema analysis problem, i.e., the detection and classification of foreign and alternative keys in PDIS.

3.1 Customizing the *Varlet Analyst*

In [15], we introduced *Generic Fuzzy Reasoning Nets* (GFRN) as an abstract, graphical language to specify heuristics and analysis processes for DBRE tools. The semantics of GFRN specifications is defined in the framework of possibilistic logic and fuzzy set theory [17]. We refer to [18] for their formal definition. The customization mechanisms of the *Varlet Analyst* are based on the GFRN approach. For this purpose, our tool provides a dedicated user interface called the *Customization Front-End* (cf. Figure 3). This user interface facilitates specification and adaption of DBRE heuristics in form of a graphical network of *predicates* (represented as ovals) and *implications* (represented as boxes). Predicates and implications are connected by arcs which are labelled by *variable names*. An arc with a black arrow head represents a logical negation. Each implication has a header with a unique implication identifier (*i1-i8* in Figure 3) and two numbers between 0 and 100 which are separated by a slash. The first number is called *confidence* and specifies a measure for the certainty that the corresponding implication is true. The second number is called *threshold* and determines the minimal amount of certainty that is required for the facts in the premise of an implication. In addition, an implication might contain a set of *constraints* over the variables of its in- and outgoing arcs.

Let us start to illustrate these concepts with implication *i2* in Figure 3. This implication represents an example for a heuristic to detect key constraints by a simple naming

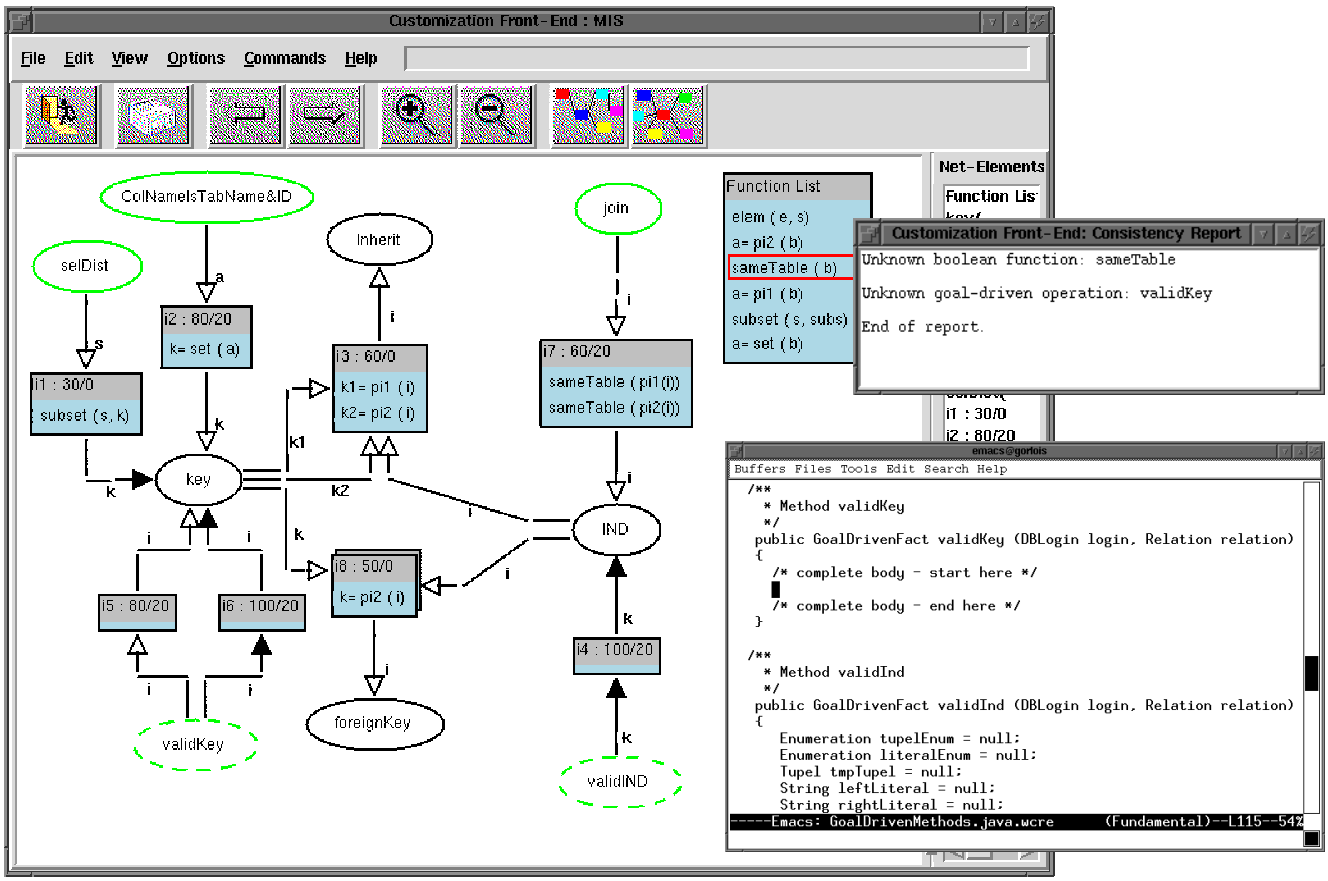

Figure 3. The *Varlet Analyst*, Customization Front-End

convention. It specifies the expectation of the reengineer that in the context of PDIS column names which are similar to their table names with the suffix "id" are credible indicators for key candidates. An example for such a situation is column *usrid* in table *USER* in Figure 4.

Indeed, this example also shows that heuristics which employ naming conventions rather deal with vague than with crisp concepts, because the name of our sample column is not exactly equal to the name of the table with suffix "id". In the GFRN approach this knowledge is represented by vague predicates, i.e., predicates that can be fulfilled partially. Figure 5 shows that string similarity measures like the *Levenshtein distance* [19] can be used to define the degrees of fulfillment for predicate *ColNameIsTabName&ID*. Figure 5 also explains the effect of the threshold: all indicators with a lower degree of fulfillment than 20 are not considered within implication *i2*.

```
create table USER(                  create table PRODGRP(
  usrid CHAR(10),                     grpname CHAR(18),
  addr CHAR(40),                      pg INTEGER,
  sname CHAR(18),                     manager CHAR(40),
  dpt CHAR(18),                       cg INTEGER)
  telo CHAR(18),
  telp CHAR(18),
  name CHAR(50))
```

Figure 4. Excerpt of PDIS schema catalog

If a column fulfills predicate *ColNameIsTabName&ID* to a degree higher than the threshold of implication *i2*, it is bound to variable *a* in the premise of *i2*. In this case, the constraint "*k=set(a)"* restricts variable *k* in the conclusion of *i2* to be a set with just one element, namely the value of variable *a*. This conversion is necessary, because, in general, key constraints are defined over *sets* of columns.

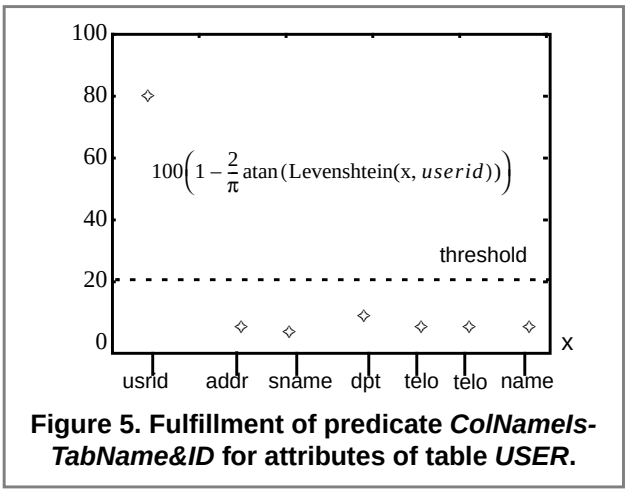

Figure 5. Fulfillment of predicate *ColNameIsTabName&ID* for attributes of table *USER*.

Obviously, an automatic analysis operation can be used to compute the degrees of fulfillment of predicate *ColNameIsTabName&ID* for all columns of a legacy DB schema. In the GFRN language such automatic analysis operations can be assigned to predicates. Depending on the point of time when these operations shall be performed, the corresponding predicates are classified as either *data-driven* or *goal-driven*. Operations assigned to data-driven predicates are performed at the *beginning* of the inference process to deliver initial information about a legacy DB, whereas operations assigned to goal-driven predicates are executed *during* the analysis process to support or refute intermediate hypotheses. In Figure 3, data-driven predicates are represented as ovals with a solid, grey border (e.g., *ColNameIsTabName&ID*), while goal-driven predicates have a dashed, grey border. All remaining predicates (e.g., *key*) are called *dependent* (black border).

A goal-driven predicate (*validKey*) is used in the premise of implication *i6*, which specifies the definite knowledge that a hypothetical key constraint has to be refuted if it is not valid in the available data of the legacy DB. This means that if the data contains at least two identical entries for this hypothetical key, then this hypotheses has to be refuted. This is modelled by the confidence of 100 for implication *i6* and the black arrow heads (which represents negation). On the other hand, implication *i5* specifies that if a hypothetical key constraint is fulfilled for a large amount of data, this fact supports the hypothesis. In this case, we cannot proof (with a confidence of 100) the key because a new entry could lead to refutation. Figure 6 shows a simplified implementation for the goal-driven analysis operation *validKey* in pseudo code.

```
algorithm validKey(T[x1,..,xk]): integer
If empty( "select * from T t1 where
          exists (select * from T t2 where
                  t1.x1=t2.x1 and ... and t1.xk=t2.xk and
                  not( t1.xk+1=t2.xk+1 and ... and t1.xn=t2.xn))")
then |T|="select count(*) from T"   /* number of entries in T */
     return 200/π*atan(|T|/100)
else return -100
```

Figure 6. Goal-driven analysis operation *validKey*

The input of the algorithm consists of the hypothetical key in form of *k* key columns $x_1,..,x_k$ of a table *T* with columns $x_1,..,x_k,..x_n$. The output of the algorithm is an integer value between 0 and 100 that depends on the total number of entries $|T|$ in table *T* if the key constraint holds (cf. Figure 7). However, if a counter-example can be found then the algorithm returns -100.

In practice, analysis operations are implemented in *Java*. The *Varlet Analyst* facilitates this task by performing consistency checks and generating code frames from corresponding predicates in the GFRN which have been

classified as data- or goal-driven (cf. the text windows in Figure 3). Furthermore, the tool provides a library of procedures which are frequently used in analysis operations.

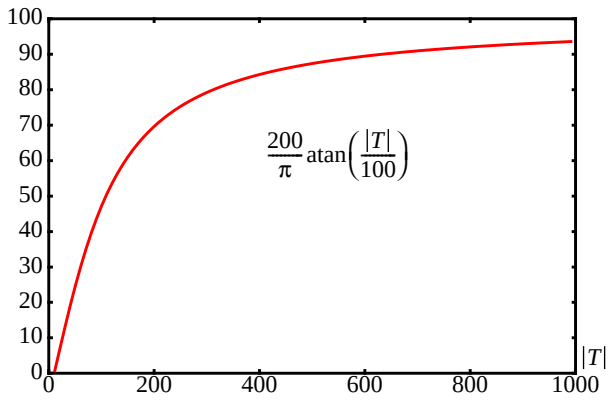

Figure 7. Result of *validkey* for the case of no counter-examples

Another valuable source of information about a legacy DB schema is the corresponding procedural code. In [20], Andersson proposes to search the code for stereotypical *code patterns* that serve as semantic indicators for schema dependencies. The GFRN in Figure 3 includes two implications (*i1* and *i7*) that deal with such code patterns. Implication *i1* in Figure 3 represents a heuristic with a confidence that highly depends on the context of a specific DBRE project. It specifies that an occurrence of a so-called *select-distinct* code pattern is an indicator against a key constraint. The left-hand side of Figure 8 shows an instance of such a pattern. This query selects entries in table *USER* according to their values in columns *sname* and *dpt*. The purpose of the SQL keyword *distinct* is to remove duplicate tuples in the result set of the query, but such duplicate tuples can only occur if columns *sname* and *dpt* do not represent a key of table *USER*. Still, by investigating some code samples during the domain analysis, the reengineer notices that the developers of PDIS frequently (mis)used the distinct keyword in their queries even if it is not needed. Consequently, the reengineer assigns a low confidence to *i1*.

```
select distinct * into :pers-rec
from USER
where sname = :SN and dpt = :DEP
```

```
select name, dpr, addr, telo
from PRODGRP g, USER u
where g.cg = :COMG
and g.manager = u.sname
```

Figure 8. Instances of stereotypical code patterns

Implication *i7* specifies the heuristic that a join between two tables might indicate an *inclusion dependency* (IND) [5]. The left-hand side of Figure 8 shows an instance for a join between tables *PRODGRP* and *USER*. In the GFRN (cf. Figure 3), a join is represented as a set of pairs of corresponding columns in the two participating tables, e.g., {(PRODUCT.pg, PRODREF.pg); (PRODUCT.cg, PRODREF.cg); (PRODUCT.no, PRODREF.prod)}. The two constraints in implication *i7* at-a-time restrict the first and the second elements of all such pairs that might constitute an IND to belong to the same table. At this, *pi1* and *pi2* represent the relational projection operation [5] on the first and the second element of each pair in variable *i*, e.g., pi1({...}) = (PRODUCT.pg, PRODUCT.cg, PRODUCT.no). The boolean function *sameTable* evaluates to true if and only if its arguments belong to the same table. The *Varlet Analyst* provides the reengineer with a list of predefined boolean and relational functions that can be used to formulate constraints for implications (cf. box *FunctionList* in Figure 3). In analogy to the integration of new analysis operations, this list can easily be extended by generating *Java* code frames for additional functions and using the provided libraries to implement their bodies.

In analogy to implication *i6*, implication *i4* specifies the definite knowledge that an hypothetical IND has to be valid in the available data. Implications *i8* and *i3* serve to classify INDs: *i3* represents the heuristic that an IND that is *key-based* and *inversely key-based* indicates an inheritance relationship (cf. [4]) , while *i8* specifies that the existence of an IND that is *key-based* indicates a foreign-key. Finally, the shadow behind the box that represents implication *i8* represents a definite implication in the opposite direction. This is shown in Figure 9 by implication *i9*. A key-based IND is a necessary condition for a given foreign key, in other words the existence of a foreignKey implies the existence of a key and a (key-based) IND.

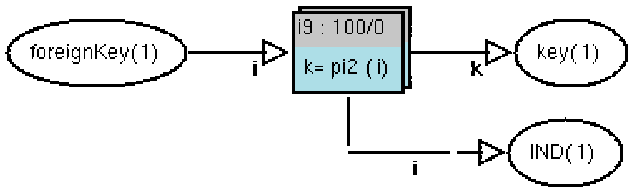

Figure 9. Opposite direction implication

3.2 Applying the *Varlet Analyst* to exploratory schema analysis

In the previous section, we exemplified the activity of customizing the *Varlet Analyst* for detecting and classifying key and foreign key dependencies in the schema of PDIS. In this section, we demonstrate the application of the *Varlet Analyst* in the actual schema analysis process. The first analysis step consists of an automatic extraction of the physical schema catalog from the used DB management system (DBMS). This reveals the structure of the participating tables including their column names and types and their primary keys or indexes (if they are specified

explicitly). If the employed DBMS is modern enough to monitor referential integrity constraints between tables (and this functionality was used by the developer of the legacy DB application) this catalog extraction might also reveal further structural information about foreign keys. Unfortunately, like in our sample scenario, these constraints are not represented explicitly in older systems. Likewise, *alternative* keys are rarely specified in the schema catalog.

Therefore, the next step in the analysis process is to invoke the data-driven analysis operations that have been specified in the GFRN during the customization process in order to detect indicators for such constraints. This step is performed automatically after extracting the schema catalog. The GFRN in Figure 3 contains three data-driven analysis operations (namely *ColNameIsTabName&ID*, *join*, *selDist*), which, applied to PDIS, deliver a set of indicators. The inference engine of the *Varlet Analyst* combines these indicators with the GFRN in Figure 3 to derive new hypotheses (e.g. *key* or *IND*) and execute goal-driven analysis operations (*validKey*, *validIND*) accordingly (cf. Figure 2). (We refer to [15] for details about its internal realization.) Subsequently, the result of this automatic analysis step is presented to the reengineer who might add situation-specific knowledge in terms of facts or hypotheses with an associated confidence value. For our application scenario, let us assume that the reengineer enters two hypotheses:

- From a conversation with a hotline operator the reengineer learns that PDIS users have a unique short name. Consequently, the reengineer adds the hypotheses that *sname* is an alternative key of table *USER* (with an assigned confidence of 70).
- Furthermore, (s)he enters his/her subjective assumption that column *pg* represents a key of table *PRODGRP* (with a confidence of 50).

Again, the inference engine executes specified goal-driven analysis operations to falsify or support these new hypotheses and the results are propagated to the current representation of the logical schema. Figure 10 shows a screenshot of the so-called *Analysis Front-End* which is used as the dialog component in the described semi-automatic schema analysis process. The logical schema is represented graphically in this dialog, where each box represents a table and INDs are visualized by lines. In order to cope with large schemas, the reengineer can choose from various levels of

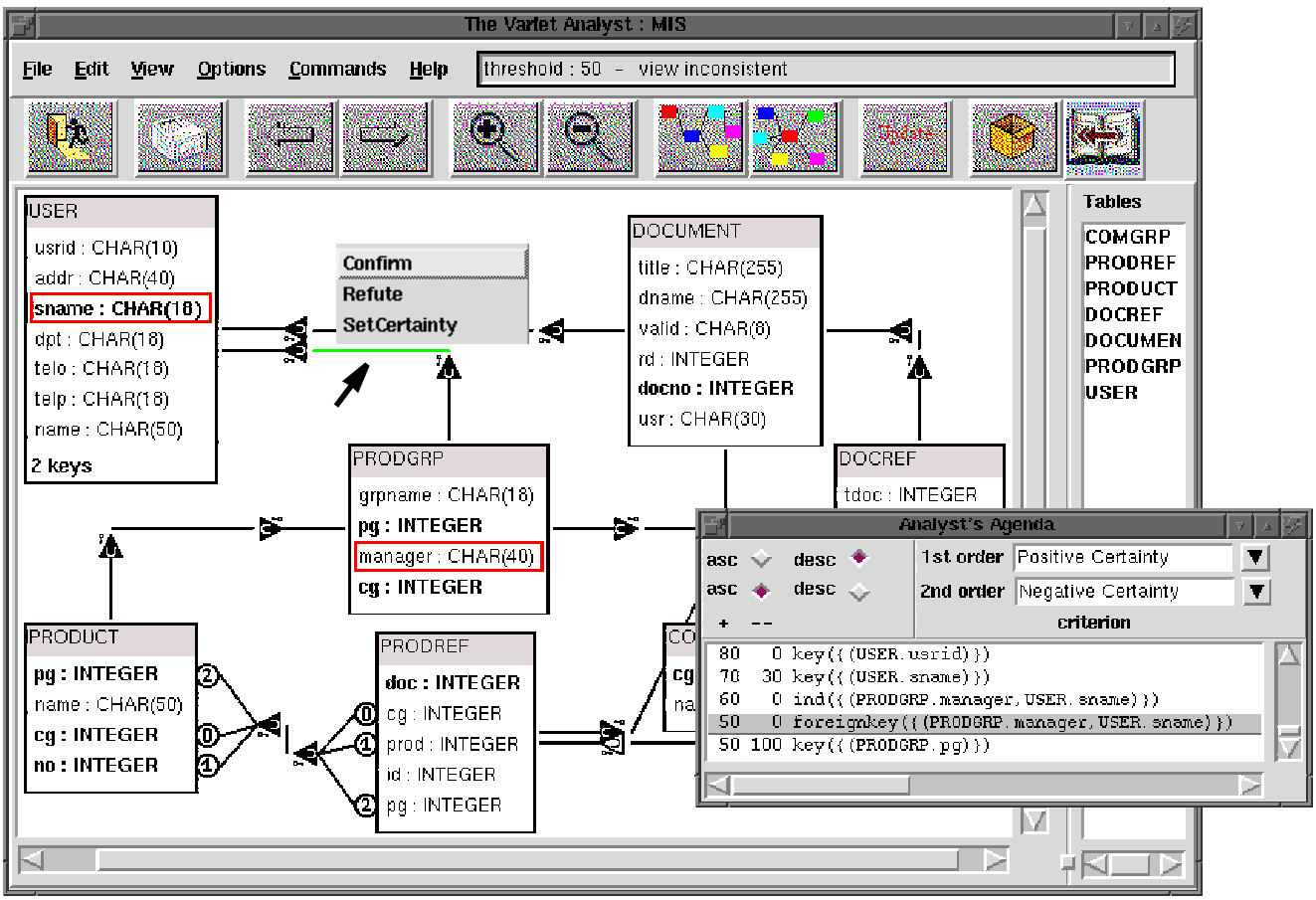

Figure 10. The *Varlet Analyst*, Analysis Front-End

abstraction and create separate views on the same logical data structure. For example, in Figure 10, most INDs are represented as single lines between tables, but the reengineer selected a detailed representation of the IND between tables *PRODUCT* and *PRODREF*. In this representation, correspondences between pairs of participating columns are marked by numbers.

A central issue that has to be tackled in a tool that exploits imperfect knowledge in DBRE is to find an adequate way to communicate such imperfect information to the reengineer and guide him/her to a consistent analysis result, incrementally. For this purpose, we have developed a dedicated dialog called the *Analyst's Agenda* which is shown in the bottom-right corner of Figure 10. The *Analyst's Agenda* presents a list of uncertain or contradicting constraints about the current view of the logical schema. For each constraint a positive and a negative confidence is displayed. The *Analyst's Agenda* provides the functionality to sort the list items according to various criterions in ascending or descending order, e.g., positive confidence, negative confidence, degree of contradiction (absolute difference of both confidences). Figure 10 shows that in our sample scenario the *Analyst's Agenda* starts with two uncertain key constraints. The first key (*usrid*) has been inferred according to the heuristic specified in the GFRN implication *i2* (cf. Figure 3), which expresses string similarity between *usrid* and *USER(ID)* (cf. Figure 5). The second key (*sname*) is listed due to the information added by the user. Both key hypotheses could not be falsified by executing the automatic goal-driven operation *validKey* (cf. Figure 6). On the contrary, the hypothesis that *sname* is a key could even be supported by the automatic data analysis. This is because table *USER* has more than 200 entries which according to the definition of operation *validKey* results in a positive confidence of 70 (cf. Figure 7). On the other hand, the analysis operation assigned to the data-driven predicate *selDist* has detected an instance of a *select-distinct* pattern that contradicts to this key assumption (cf. Figure 8). This results in a negative confidence value of 30 because in our sample GFRN the confidence of implication *i1* is limited to this value.

In addition, the *Analyst's Agenda* shows an IND from table *PRODGRP.manager* to table *USER.sname* which has been inferred due to a detected instance of a *join* pattern (cf. Figure 8) by implication *i7*. Together with the (hypothetical) key constraint for column *sname* this IND is classified as a foreign key (by implication *i8*). Note, that the subjective assumption of the reengineer (the third key constraint *PRODGRP.pg* listed in the agenda) has been falsified by the automatic data analysis operation *validKey*. This explains the negative confidence value of 100, the positive confidence value of 50 is reengineer assumption.

Now, it is up to the reengineer to investigate the intermediate results displayed by the agenda in order to confirm or refute them. If (s)he selects one of the entries listed, the corresponding schema elements are highlighted in the graphical representation. In Figure 10, the reengineer has selected the foreign key from *PRODGRP* to *USER*. After investigating the form-based user interface of PDIS (s)he confirms that the inferred foreign key in fact represents a reference between product groups and product managers (stored in table *USER*). In accordance to the GFRN, the inference engine propagates this confirmation automatically to the necessary preconditions, namely the key constraint over *sname* and the IND from *PRODGRP* to *USER*. Hence, this single confirmation causes three entries (key *USER.sname*, ind and foreignkey from *PRODGRP.manager* to *USER.sname*) to disappear from the next update of the *Analyst's Agenda*.

The entire schema analysis process consists of several iterations of such intertwined automatic and manual analysis activities until a consistent result is obtained. The inference engine that is parameterized by a GFRN mainly serves three purposes: (1) it infers new hypotheses, (2) it checks the consistency of user added hypotheses with the existing knowledge about the legacy DB, and (3) it executes automatic analysis operations to validate user input.

We would like to emphasize that the displayed positive and negative confidence values do *not* have the semantics of probabilities, but they represent *relative* measures for the compatibility or incompatibility of propositions with predicates [17]. Their sole purpose is to focus the reengineer's attention, but it is not necessary for the reengineer to understand the theory behind their inference. This theory is described in [15] and [16].

4. Architecture and implementation

The architecture of the *Varlet Analyst* is outlined in Figure 11. The entire tool comprises approximately 30 thousand lines of code. The part that deals with the internal GFRN representation and inference is written in *Java*. The implementation of the inference engine is based on the *Fuzzy Petri Net* (FPN) model which is described in detail in [15]. All boolean and relational functions that can be used in constraints of GFRN implications are implemented in module *Constraint Functions*. This module can be extended during the tool customization process if additional functions are needed (cf. Section 3.1). A predefined *Java* class *Relation* which basically implements an extended version of the relational algebra facilitates such extensions. Likewise, new analysis operations can be added to modules *Data-Driven Operations* and *Goal-Driven Operations*. Modifications to other modules are not required when new analysis operations or constraint functions are added,

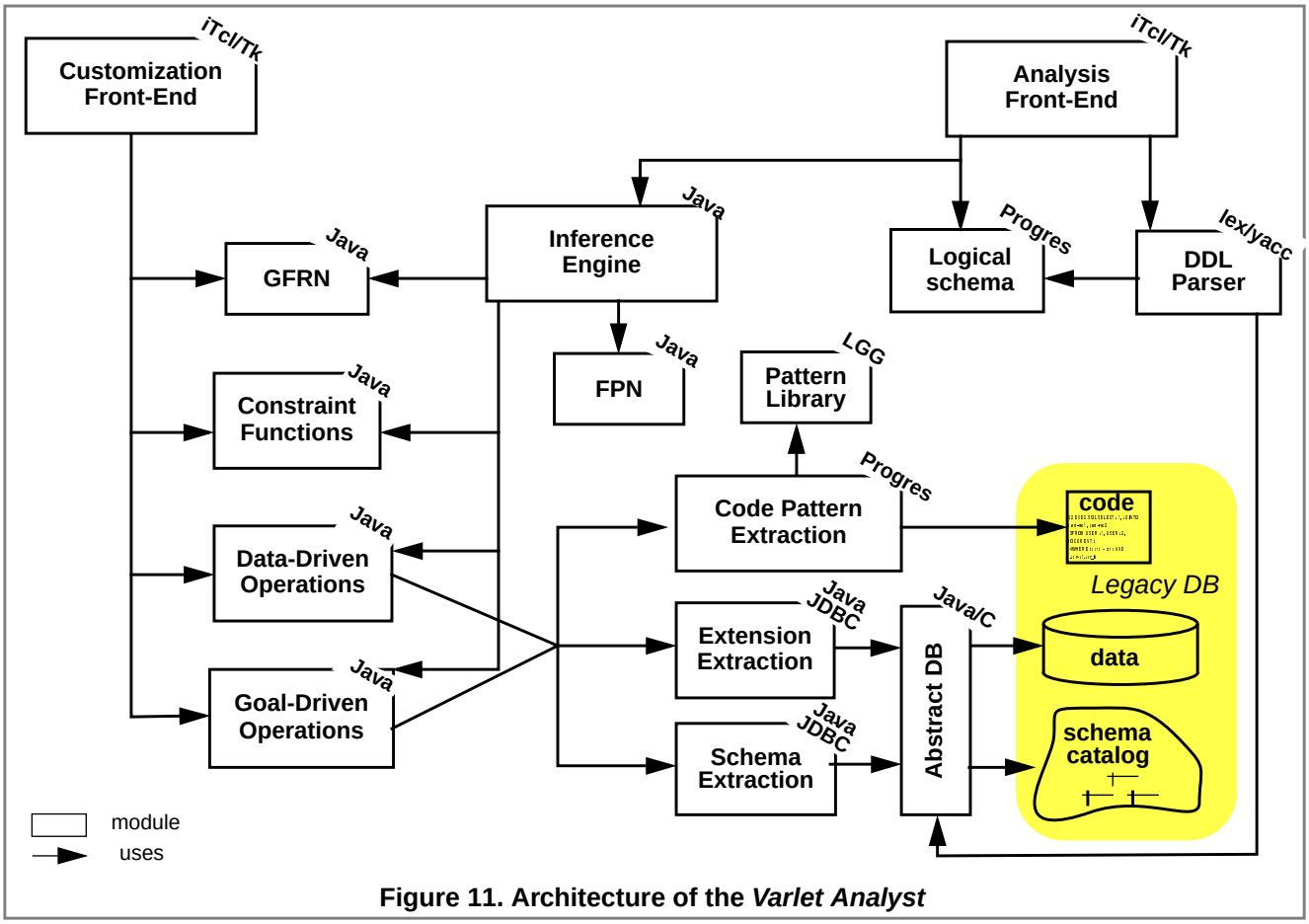

Figure 11. Architecture of the *Varlet Analyst*

because we have implemented a generic method invocation mechanism that is based on the *Java reflection API* [21]. The *Varlet Analyst* uses this API to check all available operations and functions at run-time and reports to the user in case of inconsistencies with the specified GFRN (cf. the consistency report in Figure 3).

Analysis operations use basic functionality provided by modules *Code Pattern Extraction*, *Extension Extraction*, and *Schema Extraction*. Module *Code Pattern Extraction* implements customizable detection mechanism for stereotypical code patterns. Code patterns are specified on a high level of abstraction using *layered graph grammars* (LGG) [22]. They are stored in a pattern library that can easily be extended. The actual pattern recognition algorithm is implemented in the graphical programming language *Progres* [23]. Module *Schema Extraction* provides functionality to extract information about the meta data of the legacy DB, while module *Extension Extraction* allows to access the available legacy data. We use an abstract interface to facilitate the adaption of the *Varlet Analyst* to different DB platforms.

The two user-interface components of the *Varlet Analyst*, the *Customization Front-End* and the *Analysis Front-End* have been developed in *iTcl/Tk* [24]. Internally, the logical schema is represented by an abstract syntax graph that is initially constructed by an SQL parser implemented with *lex&yacc* [25].

5. Conclusions and related work

In [26], Premerlani and Blaha emphasize that a flexible, interactive approach to DBRE is more likely to succeed than batch-oriented compilers. As a consequence, they propose a set of simple, loosely coupled tools for textual search and data analysis, e.g., *grep*, *awk*-scripts [25], and predefined DB queries. We agree that DBRE is a human-intensive and exploratory activity. However, loosely coupled tools lack the ability to control, propagate, and indicating inconsistencies. Our approach overcomes this limitation and allows to integrate such tools in terms of data-driven and goal-driven analysis operations. Still, the *Varlet Analyst* does not cut the reengineer's freedom to make manual investigations or use non-integrated tools, but it provides a basis to combine the results of such investigations.

In [12], Signore et. al. present an approach that uses *Prolog* rules to infer schema constraints from detected

semantic indicators. However, they do not consider the problem of imperfect knowledge and present inference results without any information about their confidence. Analysis operations which deliver the indicators are not explicitly considered in this approach. Furthermore, textual rules provide a much lower level of abstraction and are harder to maintain.

This paper presents a first approach to consider the mental model of the reengineer during legacy DB analysis in a DBRE tool. By tolerating and exploiting such imperfect knowledge the *Varlet Analyst* fills an important gap between the rather informal (and exploratory) activity of legacy schema analysis and the well-supported activity of schema translation, visualization, and redesign. We have applied the *Varlet Analyst* in a project with one of our industrial partners which is very similar to the case study described in this paper. The tool is well accepted because it does not impose a specific predefined sequence of analysis steps but supports the reengineer in an evolutionary process. It helps to focus the reengineer's attention on the most controversial parts of the legacy DB. In comparison with current approaches our tool plays a more active role in the analysis process. The concept of goal-driven analysis operations unburdens the reengineer from invoking such standard operations explicitly. On the contrary, the reengineer just enters his/her assumptions about the schema and the *Varlet Analyst* tries to validate them. The GFRN approach facilitates the customization of our tool with respect to changing application contexts. In particular, the concept of *thresholds* in GFRN implications provides the flexibility to adjust the *Varlet Analyst* to analyze legacy DBs of any scale: for large-scale applications, the reengineer can start the analysis with higher threshold values to reduce the set of hypotheses to the most credible ones.

Due to our experience, an understanding of the GFRN formalism or the inference algorithm is not necessary to apply the *Varlet Analyst* in the actual analysis process. Still, in some situations an explanation dialog for inferred hypotheses would be desirable. Besides the development of such a dialog, our future research activities will focus on generalizing the GFRN approach for other reengineering contexts, e.g., to recover architectural patterns [27].

Acknowledgments

We would like to thank Michael Kisker, Sergej Oseutschuk, and Felix Wolf for many fruitful discussions and their support in implementing the *Varlet Analyst*. Many thanks to Jörg Niere, Wilhelm Schäfer, Heike Schalldach, and Anke Weber for their valuable comments on this paper.

References

[1] T. J. McCabe. Does reverse engineering have a future? Keynote of the *5th Working Conference on Reverse Engineering*, Honolulu, Hawaii, USA, October 1998.

[2] R. A. Martin. Dealing with dates: Solutions for the year 2000. *Computer*, 30(3):44–51, March 1997.

[3] K. Grotenhuis. Crossing the euro rubicon. *IEEE Spectrum*, 35(10):30–33, October 1998.

[4] 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.

[5] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2nd edition, 1994.

[6] A. Behm, A. Geppert, and K. 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.

[7] Novera Software Inc., 3 Burlington Woods, Burlington, MA 01830, USA. *Novera EPIC Database Builder (TM)*, release 1.3 edition, September 1997.

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

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

[10] J.-L. Hainaut, J. Henrard, J.-M. Hick, and D. Roland. Database design recovery. *Lecture Notes in Computer Science*, 1080, pp. 272ff, 1996.

[11] 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.

[12] 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.

[13] J. H. Jahnke and A. Zündorf. *Handbook of Graph Grammars and Computing by Graph Transformation - Application*, volume 2, chapter Applying Graph Transformations To Database Re-Engineering. World Scientific, Singapore, 1999.

[14] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, May 1994.

[15] 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.

[16] 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.

[17] D. Dubois, J. Lang, and H. Prade. *Possibilistic Logic*, pages 439–503. Clarendon Press, Oxford, 1994.

[18] J. H. Jahnke. *Managing Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Dept. of Mathematics and Computer Science, 33095 Paderborn, Germany, 1999.

[19] L.I. Levenshtein. *Binary codes capable of correcting*

deletions, insertions and reversals. Soviet Physics Doklady, vol. 6, pp. 707--710, 1966.

[20] M. Andersson. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. In *Proc. of the 13th Int. Conference of the Entity Relationship Approach, Manchester*, pages 403–419. Springer, 1994.

[21] D. Flanagan. *Java in a Nutshell: a desktop quick reference*. A Nutshell handbook. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, second edition, 1997.

[22] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing, London, Academic Press.*, 8(1), 1997.

[23] A. Schürr, A. J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer, editor, *Software Engineering - ESEC '95*. Springer Verlagar Engineering with PROGRES, 1995.

[24] B. B. Welch. *Practical Programming in Tcl & Tk*. Prentice Hall Press, Upper Saddle River, 2 edition, 1997.

[25] K. Rosen, R. Rosinski, and J. Farber. *UNIX System V Release 4: An Introduction for New and Experienced Users*. Mc-Graw-Hill, New York, NY, USA, 1990.

[26] 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.

[27] J. Jahnke and A. Zundorf. Rewriting poor design patterns by good design patterns. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997. TUV-1841-97-10.