Establishing referential integrity in legacy information systems - Reality bites!

Loup Meurice
Faculty of Informatics
University of Namur, Belgium

Fco Javier Bermúdez Ruiz
Faculty of Informatics
University of Murcia, Spain

Jens H. Weber
Department of Computer Science
University of Victoria, Canada

Anthony Cleve
Faculty of Informatics
University of Namur, Belgium

***Abstract*—Most modern relational DBMS have the ability to monitor and enforce referential integrity constraints (RICs). In contrast to new applications, however, heavily evolved legacy information systems may not make use of this important feature, if their design predates its availability. The detection of RICs in legacy systems has been a long-term research topic in the DB reenginering community and a variety of different methods have been proposed, analyzing schema, application code and data. However, empirical evidence on their application for reengineering large-scale industrial systems is scarce and all too often "problems" (case studies) are carefully selected to fit a particular "solution" (method), rather than the other way around. This paper takes a different approach. We analyze in detail the issues posed in reengineering a complex, mission-critical information system to support RICs. In our analysis, we find that many of the assumptions typically made in DB reengineering methods do not readily apply. Based on our findings, we design a process and tools for detecting RICs in context of our real-world problem and provide preliminary results on their effectiveness.**

I. Introduction

Referential integrity is an important quality attribute of data stored in relational information systems (IS). It refers to the state where data stored in related tables obeys to the foreign key (FK) constraints defined between those tables. Most modern database management systems (DBMS) purposed for business applications provide features for declaring and automatically enforcing FK constraints. However, many legacy IS do not use these features - or use them only to a limited degree (i.e., for more recently developed functionality), particularly if their original design predates availability of those mechanisms in the DBMS platform. Such applications must be reengineered in order to benefit from automated integrity enforcement. Although the database layer is not the only option for enforcing RICs (presentation and application layers are other options), pushing enforcement in this layer is often seen as the most reliable option to control many concurrent "channels" entering data into the system.

From a high level perspective, this reengineering process consists of two steps, namely *FK identification* and *FK implementation*. Research activity in this area has primarily focused on the first step (identification), which can be viewed as a form of *design recovery*. A wealth of different methods and tools have been proposed to recover FKs from a variety of data sources, including the database schema, application code, data instances, and documentation. While many FK identification methods have been proposed, empirical evidence about their comparative effectiveness in real-world industrial settings remains rare.

In contrast to many other research works that start by proposing a new or improved solution to the above described reengineering problem, followed by a validation with problem case studies (often hand picked to make a point), this paper starts by studying the actual problem in context of a real-world, large-scale legacy system in the healthcare industry. As a result of our analysis, we find that many of the assumptions commonly made in DB reengineering methods and tools do not readily apply in practice. Based on our problem analysis we devise a process for reengineering legacy IS with respect to establishing referential integrity constraints, incorporating, combining and extending existing reengineering methods. We report empirical results of implementing this process in the context of our problem case study system. Our results suggest that the process of reengineering legacy IS with respect to establishing referential integrity constraints may be considerably more complex than is commonly assumed. It must be understood as an *incremental* detection process.

II. Foundations and Related work

In relational databases, referential integrity constraints (RICs) are usually declared by means of *foreign keys*. A foreign key is a simple and intuitive construct through which a row in a table is used to reference another row in another table. Considering a table S with key KS on the one hand, and a set FR of columns of table R on the other hand, if $R.FR \longrightarrow S.KS$ is declared as a foreign key, then for each row $r \in R$ (that is not null) there should exist a row s in table S such that $r.FR = s.KS$. In other words, the set of values of FR that appears in table R must be a part of the set of values of KS of table S. The foreign key FR acts as a reference to the rows of S.

Unfortunately, it is not unusual that RICs are left *implicit*, i.e., they are not explicitly declared in the DDL[1] code of the database. Several reasons can be identified: the lack of background in database design in the software development teams [1], the limitations of the target database platform, or the necessity to tolerate data inconsistencies [2]. The problem of undeclared foreign keys elicitation has thus been extensively studied and a large variety of techniques have been proposed in the last two decades, each considering a particular information system artifact as main source of information:

The first author is supported by the F.R.S.-FNRS via the DISSE project.
The second author was partially supported by the grant number 15389/PI/10 (Science and Technology Agency of the Region of Murcia, Spain).

[1]DDL stands for Data Description Language

(1) *Schema analysis* [3]–[5]. Spotting similarities in names, value domains and representative patterns may help identify hidden constructs such as foreign keys.
(2) *Data analysis* [6]–[8]. Mining the database contents can be used to discover implicit properties or to check hypothetic constructs that have been suggested by the other techniques.
(3) *GUI analysis* [9], [10]. Forms, reports and dialog boxes are user-oriented views on the database that exhibit spatial structures, meaningful names, explicit usage guidelines and, at runtime, data population and error messages that can provide information on data structures and constraints.
(4) *Static program analysis* [11]–[13]. Even simple analysis, such as dataflow graph exploration, can bring valuable information on field structure and meaningful names. More sophisticated techniques such as program slicing can be used to identify complex constraint checking or foreign keys.
(5) *Dynamic program analysis* [14]. In the case of highly dynamic program-database interactions, the database queries may only exist at runtime. Hence recent techniques allowing to capture and analyse SQL execution traces in order to retrieve, among others, implicit referential links between columns of distinct tables.

It is essential to note that none of the above techniques is generally sufficient to recover *all* implicit referential constraints: there is no formal way to prove that all the undeclared foreign keys, and only them, have been discovered through a particular technique. As often, automated analysis techniques may only *suggest* possible foreign key candidates with a certain level of confidence. In addition, the availability of a ground truth, allowing to evaluate a particular detection technique, is not a realistic working assumption in the context of large legacy systems.

III. Problem Case Study: OSCAR

Our case study is an open-source information system that is widely used in the healthcare industry in Canada, called OSCAR. OSCAR is a so-called Electronic Medical Record (EMR) system whose primary purpose is to maintain electronic patient records and interface with a variety of other information systems used in the healthcare industry. OSCAR has been developed since 2001, originally by the Department of Family Practice at McMaster University. Current development activities are coordinated by a not-for-profit company ("OSCAR EMR"), under an ISO 13485 certified development process.

OSCAR has been using MySQL as its DBMS platform. MySQL supports the choice of different alternative storage engines. During the first five years of OSCAR development, MySQL did not support a storage engine capable of enforcing referential integrity. Consequently, OSCAR's database implementation does not make significant use of FK constraints but rather consists of seemingly unrelated tables. Over the last several years, OSCAR has been migrating to MySQL's newer InnoDB storage engine, which provides full support for referential integrity enforcements. Since then, more recently developed parts of the system have made use of FK constraints. Still, the vast majority of the database tables remain without any explicit relationships in the schema. This situation has been a frequent source of frustration in the OSCAR developer community as it impedes program understanding and maintenance. It has also raised concerns with respect the integrity of patient health information and, ultimately, patient safety. Therefore, it has been a goal to reengineer OSCAR with respect to establishing more referential integrity constraints.

We encountered a number of challenges in our case study:
Size One obstacle in this process is the sheer size of the database schema. With close to five hundred tables and some of the larger tables comprising over thousands of columns, identifying FKs cannot be a manual process but requires automated tool support.
Multi-paradigm architecture Another challenge is the unevenly evolved nature of the OSCAR architecture, which uses a multitude of different paradigms to access the database. Some older application modules still use embedded (dynamic) SQL queries, while newer modules use object-relational middleware descriptors (Hibernate mapping files), and yet newer application code uses code annotation tags based on the Java Persistence Architecture (JPA) standard. Therefore, no single method for detecting FKs in application code is likely to recall all relevant relationships.
Confidential data Knowledge about the actual database instances is an important prerequisite for the process of identifying RICs. It is not uncommon that the data in legacy information systems is considered business confidential. However, patient records are among the most sensitive and highly regulated information items in any industry and they cannot commonly be made available for the purpose of software engineering, even under non-disclosure agreements. We had to create software and a process to securely encrypt the data prior to FK analysis and attain approval from the University ethics board prior to our reengineering study.

IV. Reengineering process

The reengineering process applied in this study starts with the identification of implicit integrity constraints through the *triangulation* of several RIC identification techniques. We present a process to address the RIC detection in a legacy system through the joint analysis of multiple sources of information: the database schema, the database contents and the program source code. The results obtained by each analysis technique are then combined in order to find a certain number of *likely* foreign key candidates. In the following, we describe each analysis step we follow in our reengineering process.

A. Schema Analysis

The *Schema Analysis* process is guided by the primary key constraints found in the tables of the schema. Each column $colPK$ contained in a primary key of a table is used to search for other columns in the database schema that could reference it. Algorithm 1 specifies this process. We use *tabPK* and *colPK* variables to refer to the table and the column, respectively, of the primary key side.

Algorithm 1 *Schema Analysis* algorithm

```
result ← ∅
for tabPK ∈ schema do
    for colPK ∈ tabPK.constraintPK do
        result ← result + SearchForFK(tabPK, colPK)
    end for
end for
return result
```

In the *SearchForFK* function, columns are searched based on their names and data types (SQL type, length and precision)

as we show in Algorithm 2. Variables *table* and *column* are used to refer to the table and the column, respectively, analyzed as a candidate foreign key.

Algorithm 2 *SearchForFK* function

```
procedure SearchForFK(tabPK, colPK)
    result ← ∅
    for table ∈ schema do
        for column ∈ table do
            if column ≠ colPK then
                if column.type = colPK.type
            & column.length ≥ colPK.length
            & column.precision ≥ colPK.precision
            & EqualsNames(table, column, tabPK, colPK)
              then
                    result ← result + (tabPK, colPK, table, column)
                end if
            end if
        end for
    end for
    return result
end procedure
```

The *EqualsNames* function returns *true* if the names of the columns and tables analyzed are *compatible*, and returns *false* otherwise. The meaning of *compatible* is based on the partial matching of the table and column names, according to their length. The function checks the length of the column name $colFK$ considered as foreign key candidate. If the length $\geq$ 5 characters we check whether the target table name $tabPK$ and/or the target column name $colPK$ is included in $colFK$ (only if its length is > 2, otherwise we consider that this name is not meaningful enough). If the length of $colFK$ is < 5 characters we do not check if $tabPK$ is contained in $colFK$ (because we consider that names are not meaningful) and we only check the length of $colPK$. If it is > 2, we check if $colPK$ is contained in $colFK$. Otherwise, we could suppose that $colPK$ has a name like *'id'* or something similar. In this scenario, a specific check is performed: we eliminate in $colFK$ the occurrences of $colPK$ and some other special characters like *'_'*. Then, we verify if the resulting name is part of $tabPK$. Let us illustrate this using an example. We could have a $colFK$ named *'prid'* which would be analyzed in relation to a $colPK$ named *'id'* in a table named *'provider'*. After elimination of the $colPK$ from $colFK$, we would have the string *'pr'* which would be contained in $tabPK$.

B. Data Analysis

The *Data Analysis* process utilizes the results generated by the *Schema Analysis* process as starting point. This approach is usually a necessity for large-scale legacy databases, as a brute-force data analysis with respect to detecting all potential foreign keys is usually computationally prohibitive.

Algorithm 3 *Data Analysis* algorithm

```
result ← ∅
for (tabPK.colPK, tabFK.colFK)
∈ set(tabPK.colPK, tabFK.colFK) do
    countPKReg ← select count(*) from tabPK
    countFKReg ← select count(*) from tabFK
    matching ← select colFK from tabFK
                intersect all
                select colPK from tabPK
    percentage ← (matching * 100)/countFKReg
    if percentage ≥ threshold then
        result ← result + (tabPK, colPK, tabFK, colFK)
    end if
end for
return result
```

Algorithm 3 shows how the data analysis is applied. Taking a set of foreign key candidates, the algorithm calculates the matching of values involved on each candidate. This matching defines how many values in *tabFK.colFK* can be found in *tabPK.colPK*. This matching value must be measured in relation to the number of rows in *tabFK* to calculate the percentage of matching values. The number of rows in *tabPK* is reported for better interpreting that percentage.The algorithm is set up by means of a *threshold* value which is established to only return candidate foreign keys *(tabPK, colPK, tabFK, colFK)* having a percentage value above the threshold value.

C. Static SQL Analysis

Some technologies allow one to embed SQL sentences in business logic code. For instance, the JDBC API[2] on the Java platform can embed strings (using double quotes) defining SQL sentences as parameters. One could either define *complete* SQL sentences or build such sentences using *several* fragments of SQL code. Our *Static SQL Analysis* process enables to analyze the programs source code in order to identify, parse and exploit the SQL code fragments they contain. It is a linear process composed of 5 steps, where the output of one step constitutes the input of the next step. This process has first to identify the Java files implementing the OSCAR client applications. Then, it has to parse those source code files searching for literal strings containing SQL SELECT sentences. Once these sentences have been retrieved, they are parsed to extract the FROM and WHERE clauses. A join condition in a WHERE clause must comply with the following syntax: *'columnA = columnB'*, where columns could be prefixed by table alias or table names. As future work, we will consider other ways to define a join, e.g., using the *in* operator and nested queries. We assume in our analysis process that literal strings composing an SQL sentence are adjacent and properly ordered. We need to make this assumption as we use static analysis and the program code is not interpreted by our analyzer. Below, we briefly explain how the program analysis process is carried out by executing five distinct steps:

Step 1: A parser is used to extract string literals, i.e. sequences of characters between two delimiters, in each Java file of the OSCAR system. Delimiters are special language-dependent characters, like the double-quote or the single-quote.

Step 2: We analyze the strings obtained so far and all those which are not related to a SQL SELECT statement are discarded. We assume that a string is only related to a SELECT statement if it does contain some keywords of the SQL SELECT statement syntax or some table and column names occurring in the legacy database schema.

Step 3: We re-create a complete SELECT sentence by concatenating two or more strings. To do this, information about the string in needed like: the name of the Java file where it was found, the line number in the file and the position of the first character in the line.

Step 4: We extract the FROM and WHERE clauses from the resulting SELECT sentences. A pre-parsing step deals with discarding literal strings included after Step 3 but which generate noise due to the presence of parameters that are only determined at runtime.

Step 5: We analyze the content of the WHERE clauses, searching for a join condition and resolving the identity of each column involved in it, by using the table definitions in the FROM clause.

[2]Java Database Connectivity

D. Hibernate Analysis

A large part of the OSCAR applications uses the Hibernate Object-Relational Mapping (ORM) to access the database. Hibernate allows developers to map Java classes to database tables. Those mappings are usually declared in a mapping file (an XML document) that instructs Hibernate how to map the Java classes to the database tables. We consider the Hibernate XML mapping files as another possible way to infer implicit foreign keys. Even if an ORM such as Hibernate offers an abstraction layer permitting to ignore the underlying database structures (and thus the presence of FKs), we consider that some legacy systems could use Hibernate to access to legacy databases with missing constraints (like OSCAR). Our Hibernate parser searches in each mapping file for a *'class'* tag, where an entity name is mapped to a table name by means of *'name'* and *'table'* attributes, respectively. If both names are equals, *'table'* attribute could be omitted. In a similar way, the attributes in an entity are declared by a *'property'* tag and *'name'* and *'column'* attributes. Declarations of RICs can be defined using the following tags: *'one-to-one'*, *'many-to-one'*, *'one-to-many'* and *'many-to-many'*. Different kinds of RICs are permitted in a mapping file. We illustrate below some of the most common techniques:

Many-to-one relationships In the example below, the developer defined a many-to-one constraint between the *tickler* table and the *tickler_update* table mapped to *TicklerUpdate* class. In such a case, we can infer a foreign key from table *tickler_update* to table *tickler*. Our hibernate parser identifies the *'name'* attribute as the foreign key column, and the *'class'* and *'column'* attributes as the target primary key.

```
<class name="TicklerUpdate" table="tickler_update">
<many-to-one name="tickler" class="Tickler"
column="tickler_no" update="false" insert="false" lazy="false" />
```

Many-to-many relationships In the example below, the developer defined a multi-valued association. The Hibernate parser identifies the *'name'* attribute of *'set'* tag as a foreign key column, and the *'class'* and *'column'* attributes contained in *'many-to-many'* tag as the target primary key. But in this case, since an intermediate table must be referenced for both foreign keys, the *'table'* attribute in the *'set'* tag is needed to refer to an intermediate table name.

```
<class name="Site" table="site">
<set name="providers" table="providersite" lazy="true" inverse="true">
   <key column="site_id" />
   <many-to-many column="provider_no" class="Provider" /></set>
```

SQL query declarations Hibernate also allows developers to directly use SQL queries in the mapping file. Those queries could be a good indicator for inferring RICs too, especially when the query consists of a join between two tables.

E. JPA Analysis

JPA[3] is a Java specification for persistence programming which describes the management of relational data in applications. JPA is a generic standard, independently of any particular ORM middleware. Different concrete ORM middleware products support JPA, including the aforementioned Hibernate middleware. However, using Hibernate "the JPA way" consists

[3]Java Persistence API

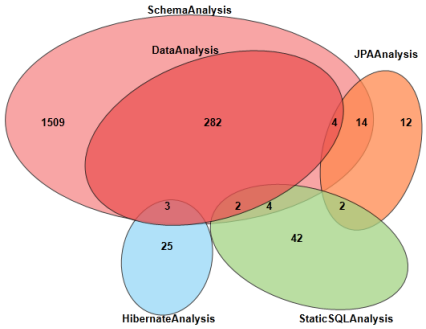

Fig. 1: Initial report

in using Java code annotations rather than XML mapping files to specify how persistent objects and their relationships are mapped to relational table structures. The most recent OSCAR components use JPA annotations rather than Hibernate mapping files. We therefore implemented a parser for JPA code annotations. For each JPA entity file a *'Table'* annotation is searched, where an entity name is mapped to a table name by means of the *'name'* attribute. If both names are equal, the *'Table'* annotation can be omitted. Declarations of RICs are defined using one of the following annotations: *'ManyToOne'*, *'OneToMany'* and *'ManyToMany'*. For instance, the following *one-to-many* annotation expresses the same RIC as in our Hibernate example. The *'JoinColumn'* annotation contains attributes to define the foreign key column (*'name'*) and the column of the target primary key (*'referencedColumnName'*). The table name containing the RIC is obtained from the entity class, and the table name referenced by the RIC is obtained from the class type in the attribute defining the relationship.

```
@Table(name = "tickler")
public class Tickler {  ...
  @OneToMany(fetch=FetchType.EAGER)
  @JoinColumn(name="tickler_no", referencedColumnName="tickler_no")
  private Set<TicklerUpdate> updates = new HashSet<TicklerUpdate>();
  ... }
```

V. Results

As described above, we have implemented 5 different techniques for recovering implicit RICs. In this Section, we present the results obtained by combining those techniques and describe our chosen strategy for accepting and rejecting the RICs identified. After applying those techniques on the OSCAR system, we extracted 1.899 FK candidates. Figure 1 illustrates the distribution through the 5 techniques: 1.818 FK candidates were detected by the *schema analysis*; 291 by the *data analysis*; 28 by the *Hibernate analysis*; 32 by the *JPA analysis*, and 50 by the *static SQL analysis*.

Another iteration was required for further exploiting those first results. We defined a list of criteria allowing us to accept a FK candidate. Each candidate FK respecting at least one of those criteria is accepted.
1. The FK is proposed by the *schema analysis* and has a *matching percentage* above or equal to 90%.
2. The FK is proposed by the *Hibernate analysis*.
3. The FK is proposed by the *JPA analysis*.
4. The FK is proposed by the *SQL static analysis* and it refers to a *primary/unique key*.

After applying those criteria, we moved from 1.899 potential to 215 accepted candidates. The 1.684 remaining ones are

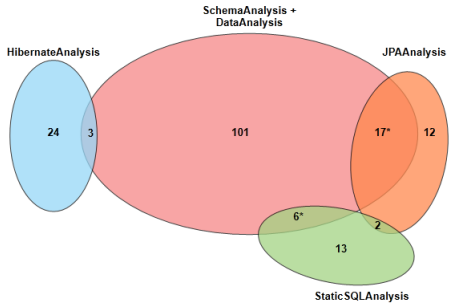

Fig. 2: Accepted FKs using our accept./reject. criteria.

considered as *unlikely*. In order to reach higher precision in our results, we also considered 4 rejection criteria.
1. *Matching* The *unlikely* candidates having a data *matching value* lower than 90% are rejected.
2. *Bi-directionality* The *unlikely* candidates such as there exists an *accepted* candidate in the opposite direction, are rejected.
3. *Unicity* The *unlikely* candidates such as there exists an accepted candidate defined on the same column(s), are rejected.
4. *Transitivity* The *accepted* candidates that could be transitively derived from other accepted candidates are rejected.

Figure 2 represents the distribution of the *accepted* candidates through the 5 information sources after having considered our rejection criteria: 146 *unlikely* candidates have been *rejected* because they do not respect the minimal matching value; 1.219 *unlikely* candidates have been *rejected* by unicity; 23 *unlikely* candidates by bi-directionality while 37 *previously-accepted* candidates have been *rejected* by transitivity.

VI. Discussion and Conclusions

Observations The results obtained when identifying FK candidates in OSCAR yield to some interesting concluding observations. First, we observe that the different sources of information have different levels of reliability. Although we do not know the actual list of implicit foreign keys that are valid in OSCAR, we can already say that the schema analysis technique may lead to overly noisy results when used in isolation. However, there is no perfect source of information that would, alone, be sufficient for identifying all implicit FKs. For instance, while the Hibernate mapping file and the JPA annotations are reliable sources of information, they allowed us to recover a very limited subset of the implicit FKs in the OSCAR schema, i.e., those involved in the most recent tables. This observation directly relates to the evolution history of the system. We saw that the management of implicit RICs in a system may be largely inconsistent over time. Some old RICs have never been explicit declared, some more recent ones have been specified through Hibernate, and some others have been declared via JPA annotations. Hence, the RIC detection approach we propose in this paper, based on the *triangulation* of several RIC identification techniques for confirming/rejecting RIC candidates, seems very promising in the context of a legacy system that has been subject to a long evolution history.

Limitations Although it has shown some merits, our multi-source FK identification process suffers from several limitations. First, the threshold of 90% for the data consistency heuristics could be further validated and calibrated with respect to the number of rows in the referencing table. In addition, since we did not have access to a ground truth, it was difficult for us to precisely quantify the reliability and the complementarity of the different identification techniques we combine. This will be a prerequisite to further improve our triangulation process and devise a more accurate FK candidate ranking method. Finally, as we mentioned above, some OSCAR tables involved in a RIC candidate being empty, our results are partially incomplete as well.

Future work We anticipate several directions for future work in the context of RIC reengineering. First, we intend to further investigate the OSCAR case study, and to involve the developers in the establishment of a ground truth, even partial. Second, we plan to consider other sources of information for the identification and ranking of RIC candidates. We think, in particular, of integrating historical information. For instance, let us assume that the history analysis reveals that the same developer has created both tables involved in a RIC candidate, this could be seen as an additional confirmation argument. In contrast, if a RIC candidate involves two very recently created tables the names of which do not appear in the Hibernate file nor in the JPA annotations, this could be considered as a rejection argument. Last but not least, we intend to devise a tool-supported methodology for assisting developers to incrementally implement identified RIC candidates in a legacy software system as well as we plan to consider using the LSD approach [15] to generalize the parsing of tags.

References

[1] M. R. Blaha and W. J. Premerlani, "Observed idiosyncracies of relational database designs," in *Proc. of WCRE'95*. IEEE CS, 1995.

[2] R. Balzer, "Tolerating inconsistency," in *Proc. of ICSE'91*. IEEE CS, 1991, pp. 158–165.

[3] S. B. Navathe and A. M. Awong, "Abstracting relational and hierarchical data with a semantic data model," in *Proc. of ER'87*, 1988, pp. 305–333.

[4] V. M. Markowitz and J. A. Makowsky, "Identifying extended entity-relationship object structures in relational schemas," *IEEE TSE*, vol. 16, no. 8, pp. 777–790, 1990.

[5] W. J. Premerlani and M. R. Blaha, "An approach for reverse engineering of relational databases," *CACM*, vol. 37, no. 5, pp. 42–ff., 1994.

[6] R. H. L. Chiang, T. M. Barron, and V. C. Storey, "Reverse engineering of relational databases: extraction of an eer model from a relational database," *Data Knowl. Eng.*, vol. 12, no. 2, pp. 107–142, 1994.

[7] S. Lopes, J.-M. Petit, and F. Toumani, "Discovering interesting inclusion dependencies: application to logical database tuning," *Inf. Syst.*, vol. 27, no. 1, pp. 1–19, 2002.

[8] H. Yao and H. J. Hamilton, "Mining functional dependencies from data," *Data Min. Knowl. Discov.*, vol. 16, no. 2, pp. 197–219, 2008.

[9] J. F. Terwilliger *et al.*, "The user interface is the conceptual model," in *Proc. of ER'06*, ser. LNCS, vol. 4215. Springer, 2006, pp. 424–436.

[10] R. Ramdoyal, A. Cleve, and J.-L. Hainaut, "Reverse engineering user interfaces for interactive database conceptual analysis," in *Proc. of CAiSE'10*, ser. LNCS, vol. 6051. Springer, 2010.

[11] J.-M. Petit *et al.*, "Using queries to improve database reverse engineering," in *Proc. of ER'94*. Springer-Verlag, 1994, pp. 369–386.

[12] G. A. Di Lucca, A. R. Fasolino, and U. de Carlini, "Recovering class diagrams from data-intensive legacy systems," in *Proc. of ICSM'00*. IEEE Computer Society, 2000, p. 52.

[13] A. Cleve, J. Henrard, and J.-L. Hainaut, "Data reverse engineering using system dependency graphs," in *Proc. of WCRE'06*. IEEE CS, 2006, pp. 157–166.

[14] A. Cleve, N. Noughi, and J.-L. Hainaut, "Dynamic program analysis for database reverse engineering," in *GTTSE*. Springer. LNCS, 2012.

[15] A. Doan, P. Domingos, and A. Halevy, "Learning to match the schemas of data sources: A multistrategy approach," *Machine Learning*, vol. 50, no. 3, pp. 279–301, 2003.