The Role of Implicit Schema Constructs in Data Quality

Anthony Cleve
PReCISE Center/LIBD
University of Namur
21 rue Grandgagnage
B-5000 Namur, Belgium
acl@info.fundp.ac.be

Jonathan Lemaitre
PReCISE Center/LIBD
University of Namur
21 rue Grandgagnage
B-5000 Namur, Belgium
jle@info.fundp.ac.be

Jean-Luc Hainaut
PReCISE Center/LIBD
University of Namur
21 rue Grandgagnage
B-5000 Namur, Belgium
jlh@info.fundp.ac.be

Christophe Mouchet
ReveR S.A.
130 Boulevard Tirou
6000 Charleroi, Belgium
cmo@rever.eu

Jean Henrard
ReveR S.A.
130 Boulevard Tirou
6000 Charleroi, Belgium
jhe@rever.eu

ABSTRACT

This paper presents a comprehensive approach to legacy data quality assessment and improvement. It is based on an initial database reverse engineering phase that allows to recover *implicit* constructs, that is, structures and constraints that have not been explicitly declared in the database schema, using program analysis techniques. These constructs then serve as a basis for detecting data inconsistencies, identifying unsafe program fragments, and proposing necessary improvements at both the database and program sides.

1. INTRODUCTION

Information Quality (IQ) is now considered as an important research field as much for the industries exploiting data as for research groups considering it as an important challenge. For industry IQ mainly has an economical impact underlined by authors through financial cost [23]. As a research challenge many definitions and methods have been proposed so far. Some of them may be considered as strongly accepted as those given by Ballou and Pazer [1], Wang and Wand [26] or Wang and Strong [27] that are the basis of the current definitions of data quality. Generally accepted terms for data quality include *accuracy*, *completeness*, *consistency* and *currency* [18]. In this context many researchers have proposed and still propose new approaches to:

- detecting erroneous data in a database;
- consistently querying a database containing wrong data;
- cleaning a database, by correcting or discarding data errors.

These works have been conducted in various research contexts like database integration, database migration, data warehouse feeding or reverse engineering. The same holds for the technological space that comprises traditional relational databases, object-relational databases, XML data repository, legacy systems (e.g., CODASYL DBMSs and IMS), standard files, etc.

Related works mainly concentrate on the definition of data quality and on the data cleaning process. According to the existing quality terminology this work addresses the data consistency problem. The latter generally reflects situations where different values of the same concept are present in the database, or where some data instances violate particular rules. In the 90's Wang and Wand [26] described data consistency in the following terms: *"a data value can only be expected to be the same for the same situation"*. More recently Scannapieco et al. [24] proposed a more precise definition: *"The consistency dimension captures the violation of semantic rules defined over (a set of) data items. With reference to the relational theory, integrity constraints are an instantiation of such semantic rules"*. Using a semiotic approach, Price and Shanks [20] used the concept of *"conforming to rules"* instead of the term *"consistency"* in their framework. The concept represents the fact that *"the data obeys business and other integrity rules"*. A lot of methods have been proposed in the purpose of detecting and resolving data defects [21]. The related literature comprises dedicated frameworks [11, 6, 18, 17], particular methods using data properties [10], statistical analysis techniques [7] as well as solutions based on heuristics [5].

While recent results on data quality mainly take the data themselves as inputs for detecting integrity problems, the contribution of our work is to also consider both the underlying database schema and the application programs. Moreover, we concentrate on the *intended schema*, that is, the physical schema as it is declared in the DDL code, augmented with *implicit constructs* recovered through reverse engineering techniques. Analyzing the data access behavior of the programs allows us to identify integrity constraints that are not explicitly declared in the DDL code. These additional constraints then serve as a basis for (1) assessing the quality of the data, (2) identifying unsafe critical code fragments, and (3) proposing adequate system improvements. The principles of the approach are illustrated in the context of legacy systems developed on top of relational databases. A real-world example of *CODASYL-to-DB2 migration* will then be discussed.

This paper is structured as follows: Section 2 describes in more detail the technical context of the inconsistencies. Section 3 presents a proposal for detecting implicit constraints and associated data inconsistencies. Section 4 elaborates on possible solutions for improving data filtering at both

database and program sides. In Section 5 we report on a tool-supported industrial experience for which we used the approach described in this paper. Our concluding remarks and research perspectives are presented in Section 6.

2. PROBLEM DESCRIPTION

As explained above, we focus the discussion on a particular kind of consistency problems: conflicting data against schema constructs (that is, structures and/or constraints). In this section, we further describe the problematic situation motivating our work as well as the concepts we will use in this paper.

2.1 The Problem of Data Validation

The formal rules defining data consistency may be associated to data structures or integrity constraints of various categories. Among them, let us mention:

- basic integrity constraints: primary keys, functional dependencies, cardinalities, types;
- referential constraints: foreign keys, inclusion constraint;
- existence constraints: exclusive (A and B cannot be not null at the same time), coexistence (A and B must be null or not null at the same time), implication (if B is not null, then A must not null either;
- domain constraints: interval of values, null values;
- derived values and intended redundancies.

We distinguish two main approaches to managing integrity constraints, namely database (DB) side data validation (Section 2.2) and program side data validation (Section 2.3).

2.2 Database Side Validation

According to the database side validation approach, the integrity constraints are managed by the database management system. Database side constraints may be of two forms. First, they can be declared by means of *native constructs* in the DDL code. Typical examples are primary keys, unique predicates , foreign keys, data type, not null columns and format constraints. The second category of constraints are specified by means of programmed SQL components such as check predicates, triggers and stored procedures.

The correct use of both validation solutions greatly reduces the risk of introducing inconsistencies in the database. Indeed, whatever the path used for modifying the data (direct access, programs, middleware, wrappers, ...) the specified constraints will always be verified by the DBMS, that will reject any modification request that attempts to violate constraints.

2.3 Program Side Validation

Integrity constraints validation may also be implemented within application programs. Different policies exist depending on the level of DB access centralization of the underlying system architecture:

- *Centralized management* relies on the use of data access component(s). Such a component, also known as *data wrapper* [19], provides the application programs with a database manipulation API (reading, insertion, modification). It is also in charge of data consistency management, which consists in verifying that each requested database operation does not violate corresponding constraints.
- *Distributed management* captures the situation according to which data validation is scattered over the various application programs. Each program manipulating the database should verify the constraints that its behavior could potentially violate, which correspond to constraints that are not checked at the DB side. This pattern defines the worst architecture as far as integrity control is concerned, and, quite naturally, is the most popular.

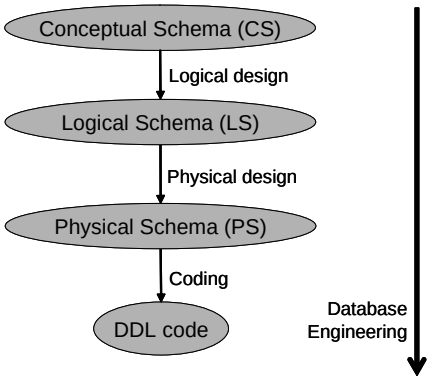

Figure 1: Standard database engineering process.

We distinguish two main data validation strategies at the program side: *reactive validation* and *proactive validation*.

- *Reactive validation* verification tasks are performed before executing a database modification that could challenge data consistency. For instance, before inserting a new order the program verifies that the corresponding customer already exist in the database.
- *Proactive validation* consists in enforcing data integrity rules during the query construction process itself, in such a way that the executed database operations never violate integrity constraints. This kind of behavior is typically implemented using user interface restrictions. For example, when encoding a new room reservation the user must select the room identifier among the list of available room references.

2.4 Database Engineering in an Ideal World

The process of designing and implementing a database that has to meet specific user requirements has been described extensively in the literature [3] and has been available for several decades in standard methodologies and CASE tools. It is made up of four main subprocesses, each producing a database schema at different levels of abstraction:

(1) *Conceptual design* is intended to translate user requirements into a conceptual schema, which is a platform-independent abstract specification of the future database. This schema collects all the information structures and constraints of interest.

(2) *Logical design*, which produces an operational logical schema that translates the constructs of the conceptual

schema according to a specific technology family without loss of semantics. The transformational approach to database engineering ([13]) allows this process to be automated.

(3) *Physical design*, which augments the logical schema with performance-oriented constructs and parameters, such as indexes, buffer management strategies or lock management policies.

(4) *Coding*, which translates the physical schema (and some other artifacts) into the DDL (*Data Definition Language*) code of the database management system. Structural DDL declaration code SQL as well as SQL components such as checks, triggers and stored procedures are generated to code the information structures and constraints of the physical schema.

Figure 1 illustrates this ideal view of database engineering, according to which each construct expressed in the conceptual schema is (correctly) implemented into SQL objects. Let $C(s)$ denote the set of constructs expressed in a schema s. Then an ideal DDL code would be such that $C(DDL) = C(CS)$.

2.5 The Concept of Implicit Constructs

Unfortunately, many databases have not been developed in a disciplined way, that is, from a preliminary conceptual schema. This was true for old systems, but loose empirical design approaches keep being widespread for modern databases due, notably, to time constraints, poor database education and the increasing use of object-oriented middleware that tend to consider the database as the mere implementation of persistent classes. Secondly, the logical (DBMS-dependent) schema, that is supposed to be derived from the conceptual schema and to translate all its semantics, generally misses several conceptual constructs. This is due to several reasons, among which the poor expressive power of DBMS models, the lazziness, awkwardness or illiteracy of some programmers [4], or the need for performance.

From all this, it results that the logical schema often is incomplete and that the DDL code that expresses the DBMS schema in physical constructs ignores important structures and properties of the data. The missing constructs are called *implicit constructs* ($ImpC$), in contrast with the *explicit constructs* ($ExpC$) that are declared in the DDL of the DBMS.

Several field experiments and projects have shown that as much as half of the semantics of the data structures are implicit [15]. Therefore, merely parsing the DDL code of the database, or, equivalently, extracting the physical schema from the system tables, sometimes provides barely half the actual data structures and integrity constraints of the database. Hence the need for database reverse engineering [12], the goal of which is to recover the complete conceptual schema of a legacy database. In particular, it aims at identifying implicit constructs by analyzing programs source code, database procedural components, as well as other artefacts like screens, reports, or programmer guides (see Figure 2).

Among the set of $ImpC$ we identify two categories based on the component(s) in charge of managing the constructs:

- $ImpC_D$ is the set of implicit constructs that are verified at the database side;

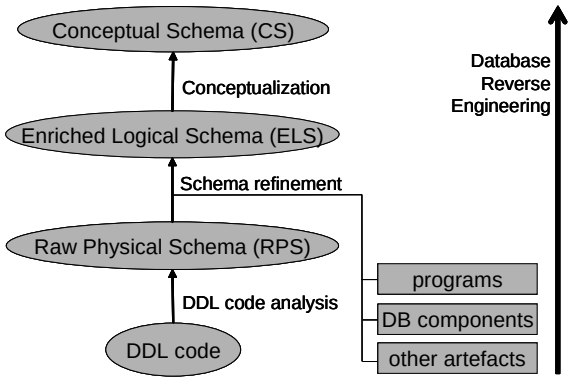

Figure 2: Implicit constructs recovery during data reverse engineering.

- $ImpC_P$ is the set of implicit constructs that are checked at the program side.

We can now define the set of implicit constructs as the difference between the conceptual schema and the DDL code: $ImpC = C(CS) - C(DDL)$. Ideally, it should hold that $ImpC = ImpC_D \cup ImpC_P$, which means that each implicit construct is managed somewhere in the system. Note that it could happen that some constructs are validated at both the database and program sides, which translates as $ImpC_D \cap ImpC_P \neq \emptyset$.

3. ERROR DETECTION

The objective of the error detection phase is to identify data errors due to the violation of implicit constructs. This phase involves two successive problems:

1. recovering implicit constructs, and more specifically $ImpC_P$;
2. detecting data inconsistencies against those constructs.

3.1 Retrieving Implicit Constructs

The implicit constructs recovery process takes as main input the source code of the application programs. Our approach relies on the use of dataflow analysis techniques [8] applied to the source code, with a focus on database operations. The final goal of the analysis is to observe the way potential implicit constructs are *used* or *checked* by the application programs.

Our dataflow analysis approach is based on the concept of *System Dependency Graph* (SDG), introduced by Horwitz and al.[16]. The SDG for program P is a directed graph whose nodes are connected by several kinds of arcs. The nodes represent assignment statements, control predicates, procedure calls and parameters passed to and from procedures (on the calling side and in the called procedure).

The arcs translates dependencies among program components. An arc represents either a *control dependency* or a *data dependency*. A control dependency arc from node v_1 to node v_2 means that, during execution, v_2 can be executed/evaluated only if v_1 has been executed/evaluated[1]. Intuitively, a data dependency arc from node v_1 to node

[1]The definition is slightly different for calling arcs, but this does not change the principle.

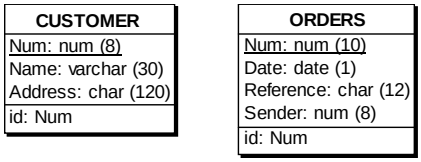

Figure 3: Two tables including implicit constructs

```
select substring(Address from 61 for 30), Reference
into   :CITY, :PRODUCT
from   CUSTOMER C, ORDERS O
where  C.Num = O.Sender
and    O.Num = :ORDID
```

Figure 4: Extracting hidden City and Product information (predicative join)

v_2 means that the state of objects used in v_2 can be defined/changed by the evaluation of v_1.

In order to illustrate the use of dataflow analysis for database reverse engineering, let us consider the small but representative example based on the schema of Figure 3. This schema is made up of two tables describing customers and orders. It translates the constructs declared in the DDL code. The SQL query of Figure 4 obviously extracts the customer city and the ordered product for a definite order. It asks the DBMS to extract these data from the row built by joining tables `CUSTOMER` and `ORDERS` for that order. It exhibits the main features of the program/query interface: the program transmits an input value through host variable `ORDID` and the query transmits result values in host variables `CITY` and `PRODUCT`. The analysis of this query brings to light some important hidden information:

1. The join is performed on columns `Num` and `Sender`. The former is the primary key (the main identifying column) of table `CUSTOMER` while the second one plays no role so far. Now, we know that most joins found in application programs are based on the matching of a foreign key and a primary key. As a consequence, this source code fragment strongly suggests that column `Sender` is a foreign key to table `CUSTOMER`. Further analysis will confirm or reject this hypothesis.

2. The seemingly atomic column `Address` appears to actually be a compound field, since its substring at positions 61 to 90 is extracted and stored into a variable named `CITY`.

Translating this new knowledge in the original schema leads to the more precise schema of Figure 5.

Obviously, the implicit constructs recovered by program analyzers are only *candidate* integrity rules, that still have to be validated. This can be done via user validation or data analysis. Figure 6 depicts diagnosis box of the dependency validation tool we have developed as a plug-in of the DB-MAIN CASE tool, which allows the user to visualise the set of database fields dependencies detected by dataflow analysis. For each resulting dependency, the tool provides the user with (1) the source and target database fields or columns, (2) their corresponding record types or tables, (3) a source code fragment (also known as *program slice*) from which the

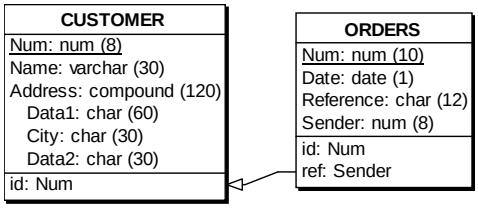

Figure 5: Potential implicit constructs revealed by source code analysis

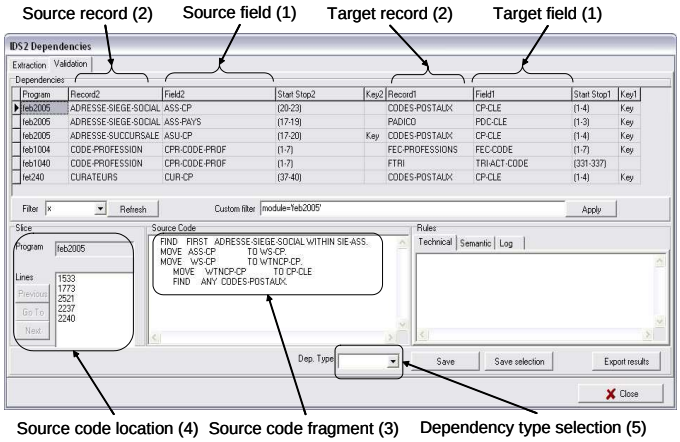

Figure 6: Result of data dependency analysis to be validated

dataflow has been derived, and (4) the corresponding source code location (program name and lines of code). Based on these results, the user may validate or discard the detected dependencies between fields. He can qualify each dependency as *no dependency*, *foreign key* or *redundancy* (5).

3.2 Detecting Data Errors

Once the implicit constraints have been validated they can serve as a basis for assessing the quality of the data themselves, by detecting inconsistent data against recovered constraints. The approach we have developed consists in automatically generating database queries looking for violation of the implicit constraints specified in the enriched logical schema. As an example, the following query will be produced from the implicit referential constraint of Figure 5:

```
select *
from ORDERS
where Sender not in (select Num
                     from CUSTOMER)
```

This query retrieves all the instances of table `ORDERS` that do not reference an existing row of table `CUSTOMER`.

4. QUALITY IMPROVEMENT

The goal of the system improvement phase is twofold. Firstly, the database must be made consistent with respect to the implicit constructs that have been recovered. This data cleansing process involves the correction/deletion/isolation of the erroneous data identified during the previous step. Secondly, we have to make sure that such data inconsistencies will to occur in the future. This necessitates to check that each implicit constraint is correctly and fully managed either at the database side or at the program side.

4.1 Data Correction

Data correction approaches can be divided into three categories [11]: manual, semi-automatic and automatic. Manual correction is generally the less cost effective solution but is the more accurate (using competent people). Automatic correction requires a lower cost but relies on predefined strategy based on rules and heuristics. So the accuracy may decline depending of the context. Semi-automatic solutions represent a balanced compromise between manual and fully-automated in terms of cost and accuracy. Semi-automatic data correction seems to be the most frequent scenario. A threshold may be defined so that when a confidence value fall under the threshold then the values have to be fixed manually [11] otherwise the correction is done automatically. Another example is the knowledge-based framework developed by Low et al. [17] working as an expert system for data correction. Finally, the interactive system proposed by Raman and Hellerstein [22] allows the user to gradually build the correction plans.

In the present work, data correction itself is rather seen as a business problem. Indeed, although automated analysis may allow to detect data consistency problems, fixing these problems often remains under the responsability of the business. For instance, let us suppose that data analysis detects a set of `ORDERS` with an invalid value of `Sender`. From a technical point of view, several correction strategies are possible: discarding the orders, correcting the orders, introducing new customers, or relaxing the referential constraint. But choosing between these solutions is rather a business decision.

4.2 Improving DB side Data Validation

Improving data filtering at the DB side can be done by introducing additional rules corresponding to the recovered implicit constraints. Such rules can be expressed within various kinds of components like new native constraints (unique, foreign key), checks, triggers or stored procedures.

Although most of the additional code fragments can be automatically generated, in practice their development may also necessitate human decisions. This holds, for instance, for the delete/update mode selection in the case of referential constraints (*cascade*, *no-action*, *set default* or *set null*).

Some data filtering improvements may necessitate the migration of the data instances involving a possibly complex ETL process. However, other improvements may have a negative impact on the successful execution of existing programs. In some situations, the use of a fault-tolerant filtering strategy may be preferable. According to such a strategy data errors may still be introduced but the filtering component systematically reports on those errors. Error reports then serve as a basis for (1) correcting erroneous data instances, (2) identifying the sources of errors in the application programs or the underlying business processes and (3) proposing necessary improvements. This idea of temporarily tolerating inconsistency has been proposed by Balzer [2].

4.3 Improving Program side Validation

The second improvement strategy concerns the application programs and particularly focuses on their data validation code fragments. It aims at identifying and fixing the unsafe data access paths that could (have) lead to the insertion of inconsistent data with respect to implicit constraints.

In theory, the set of implicit constraints that are considered during program side validation are the constraints that are not managed at the DB side (after DB side filtering improvement). In practice, some constraints that are already managed by the DBMS may also be considered in order to verify that they are not also checked at the program side. Such double validations should be avoided for obvious performance reasons.

4.3.1 Identifying Unsafe Data Access Paths

The identification of unsafe access paths is quite similar to the implicit constraints recovery process. However, it slightly differs from the latter in that it focuses on a subset of database operations accessing a subset of database entity types. Indeed, its goal is to analyze *critical database operations*. A critical database operation with respect to a constraint c is a database modification instruction that could potentially violate c. An *unsafe data access path* with respect to an implicit constraint c is a source code fragment that includes at least one critical database operation for c and that does not (fully) prevent c to be violated.

A typical example is the implicit referential constraint. Let us consider again the implicit foreign key Sender (denoted by R) from table `ORDERS` to table `CUSTOMER`. The following database operations are critical for R:

- `update CUSTOMER`, where the primary key value is modified;
- `delete from CUSTOMER`;
- `insert into ORDERS`;
- `update ORDERS`, where foreign key value is modified.

Each program executing one of these operations must make sure to preserve data consistency against R. This verification should execute a validation query selecting rows from table `CUSTOMER` (resp. `ORDERS`) for operations on table `ORDERS` (resp. `CUSTOMER`). In that particular context, detecting unsafe data access paths could be done by looking for source code locations where critical operations described above are not preceded by a suitable verification query, or that do not behave correctly according to the result of such a query.

Figure 7 gives an example pseudo-code fragment involving a correctly validated critical operation. Before inserting a new row in table `ORDERS`, the program checks that the value of variable `NewSender` corresponds to an existing customer reference. This example corresponds to a typical code pattern of *reactive* program-side validation. According to this code pattern, the dependency between the critical operation and the verification query is twofold. First, the critical operation will be executed if and only if the result of the verification query is satisfying. Second, the input arguments of the verification query are included as input arguments of the modification query. In other words, there exists a data dependency relationship between the inputs variables of the validation query and a subset of the input variables of the critical operation (variable `NewSender` in our example).

In the case of proactive validation of an implicit referential constraint, the inter-query dependency is of another nature. Indeed, the objective of the validation query is rather to retrieve the set of valid foreign key values (i.e., the set of existing primary keys in the target table). The user must then select one value among this set. The insert operation

```
exec sql
  select count(*) into :NBR
  from CUSTOMER
  where CustNum = :NewSender
end-exec
if (NBR = 0)
   display('unknown customer - insertion aborted !')
else
   exec sql
     insert into ORDERS(Num, Date, Reference, Sender)
     values(:NewNum, :NewDate, :NewRef, :NewSender)
   end-exec
end-if
```

Figure 7: Sample code fragment with implicit referential constraint validation

then uses this value as input parameter. Thus, in this case, there exists a dataflow relationship between the output variables of the validation query and the input variables of the critical operation.

4.3.2 *Fixing Unsafe Data Access Paths*

When unvalidated critical operations are identified, different solutions can be considered. The most obvious correction consists of direct code insertion. The program source code is adapted such that each critical operation requires a corresponding validation step.

There typically exists several solutions. In our example of implicit foreign key, an unsafe delete statement on table `CUSTOMER` can be transformed in different ways. First, we can add a verification query, that checks if the customer to discard still has orders in the database. If this is the case, the deletion is aborted (i.e., no action mode). Another adaptation could be to systematically delete all associated `ORDER` rows before deleting each `CUSTOMER` row (i.e., cascade mode). In both cases, the program adaptation can be automated.

Another program improvement strategy consists in reorganizing the data manipulation code fragments by introducing an additional layer, which provides the application programs with a database access API. This data access module, or *wrappers*, a large part of which can be generated, is in charge of managing data consistency when performing critical operations. The legacy program code can be interfaced with this new component with minimal adaptation. Such a source code refactoring allows to better modularize, and therefore to better master the data consistency management concern. This approach has been reported in [25].

5. INDUSTRIAL APPLICATION

The approach and the tools that support it have been recently used in the context of an industrial database migration project, conducted by ReVeR S.A. The goal of this project was to migrate a CODASYL database (IDS/II) towards a relational platform (DB2). The legacy system is in use in a Belgian federal administration. More information on our migration approach itself can be found in [14, 9].

The physical schema comprises 42 areas, 112 record types, 73 sets (relationship types), 1249 fields. A database reverse engineering phase allows us to recover the complete logical schema including implicit constraints (among which 14 additional foreign keys), more expressive names, and finer-grained field decompositions (a total of 1511 fields among which 655 are optional).

The target DB2 schema include 147 tables, 1843 columns and a total of 144 foreign keys. A subset of the foreign keys have not been explicitly declared in the DDL code. They are rather managed by fault-tolerant triggers that produce a warning in a dedicated log table each time the relational constraint is violated by a database operation. These triggers were automatically generated. This approach allows to better master the referential constraints while avoiding to disturb the execution of legacy applications.

5.1 Quality Analysis Process

In the context of database migration, the problem of data quality analysis is of primary importance, a data cleaning process being often necessary. Indeed, the migrated data must respect the DDL constraints of the target database in order to be succesfully migrated. More generally, data migration appears as an excellent opportunity to perform some data cleaning as well as to apply some improvements at the program level.

During the data error detection phase, we particularly focused on format constraints (e.g., dates, zip codes, optional and default values) and implicit foreign keys. This phase involved the following steps:

1. The legacy IDS/II data were loaded into an intermediate DB2 database, in which each column is of type string, allowing error-free data migration. The DB2 database contained more than 415 millions of rows.
2. The underlying database schema was enriched with the following annotations:
   - The expected format of each DB2 column;
   - The validity constraints associated to each column (expressed as SQL *where* clauses);
   - The implicit foreign keys between tables.
3. Starting from this annotated schema, a data analysis program generated SQL queries for inspecting the intermediate DB2 database. This analysis produced an html document reporting on the detected data errors.

Figure 8 depicts an excerpt of this report. For each table, the report details the associated format constraints that are satisfied or violated by the data. For each violated constraint, it indicates the number of problematic instances and allows the user to browse the inconsistent records (via the 'Details' link). A separate part of the report is dedicated to foreign key inconsistencies. In this project, 76 implicit foreign keys have been considered, among which 24 proved to be violated by the legacy data. Regarding data format, a total of 3497 SQL queries were executed, allowing the detection of errors in almost 70% of the tables.

A typical problem was the presence of invalid dates. In the legacy database, date fields are represented as numeric values of the form 'YYYYMMDD'. In the target DB2 database, they are expressed as columns of SQL type Date. We encountered a large amount of inconsistent dates like '20070931'. This is mainly due to the behavior of some application programs considering day '31' as the last day of the current month, whatever the month. We also encountered special date values like '00000000' or '99999999' that are used by programs for simulating the *null* value and a future date, respectively. In total, around 35 millions of invalid date values were found.

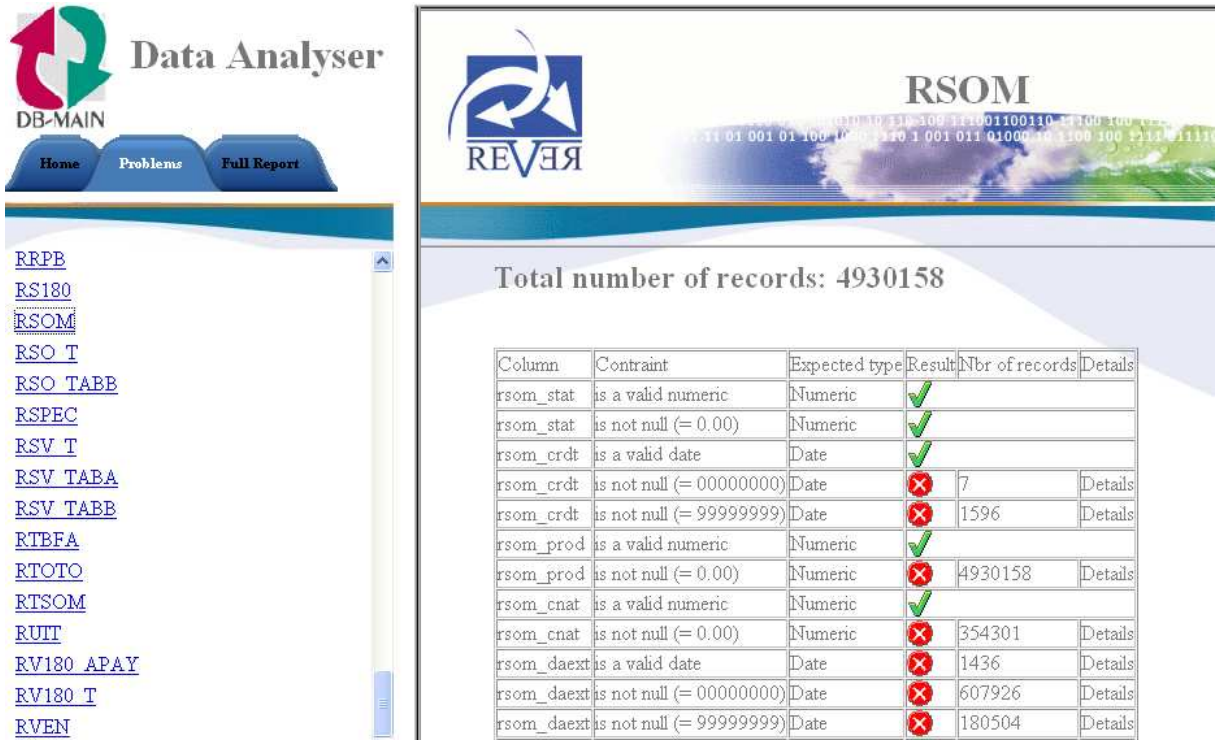

Figure 8: Excerpt of the error report produced by the data analyser

5.2 System Improvements

Based on the error report, the customer invested one complete week for (1) improving the quality of the data and (2) adapting the most problematic application programs. The latter improvements included the correction of the '*last day of the month*' problem, and the standardization of default values (e.g., a date '00000000' becomes *null*). As expected, a subset of erroneous fields could not be fixed, which required to relax some constraints. For instance, a subset of fields representing invalid dates had to be migrated as numeric columns. Most of the (still) implicit constraints are now managed by dedicated triggers introduced at the DB side. These triggers systematically produce warnings when new data inconsistencies are inserted in the database.

5.3 Lessons Learned

This industrial project showed us that completely cleaning the entire database was not achievable, especially because the migrated data actually contains legal information that cannot be modified. Moreover, a subset of the data originates from external administrative entities which complicates data correction. In such situations, the only realistic solution consists in tolerating inconsistencies by temporarily relaxing some constraints.

The IT management people at the customer side did react very positively to our results. They spontaneously decided to invest a significant effort in the correction of most data errors. They also intend to undertake further adjustments to the application programs in the next future.

Another important conclusion is that assessing the quality of constantly evolving data is more than challenging. Ideally the database should be analysed everyday, which is not feasible in practice. We also learned that while data analysis may serve as a basis for validating candidate integrity constraints, it can also contribute to the identification of new implicit constraints. In the case of this project, detecting special date values for a given column suggests that the latter should be declared optional.

6. CONCLUSIONS

The paper describes a framework for data evaluation, data correction and program improvement based on implicit construct violation. This data defect source is both very important and prone to automated processing. The comprehensive framework described in this paper encompasses the data error identification process, but also the identification of the source of the errors in application programs (the *unsafe data paths*) and the correction of the latter. Its main contributions are the complete treatment of this important source of errors thanks to now mature reverse engineering techniques [12], the use of program analysis and transformation techniques, and the improvement of the whole application system, covering data, schemas and programs. In addition, the framework is tolerant to practical constraints such as the preservation of programs whose execution relies on data violation. The approach is supported by a suite of analysis, reporting, generation and transformation tools and has been successfully applied on a large administrative system migration.

It is important to note that the framework is fully generic, and can be applied to any data source, whatever its underlying data model. It has been experimented on an IDS/II to DB2 migration, but most of its principles and tools are platform-independent and can be quickly customized to any technology. In particular, being based on the DB-MAIN

database engineering approaches and tools, it can profit from its generic meta-model and model-independent transformation technology [13].

Beyond tool refinement and further validation through additional industrial projects, an important way of improvement of the approach consists in applying learning techniques to increase the automation part of the program correction process. Indeed, most program corrections obey to systematic parametrized patterns. Learning from these heuristics, an intelligent correction help system can be designed to anticipate the kind of modification that should be applied in each case. Such an approach has been used in data correction ([22]) and can be applied to program correction as well. We plan to work in this direction.

7. REFERENCES

[1] D. P. Ballou and H. L. Pazer. Modeling data and process quality in multi-input, multi-output information systems. *Management Science*, 31(2):150–162, feb 1985.

[2] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th international conference on Software engineering (ICSE'91)*, pages 158–165, 1991.

[3] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual database design: an Entity-relationship approach*. Benjamin-Cummings Publishing Co., Inc., 1992.

[4] M. R. Blaha and W. J. Premerlani. Observed idiosyncracies of relational database designs. In *Proc. of the Second Working Conference on Reverse Engineering (WCRE'95)*, page 116, Washington, DC, USA, 1995. IEEE Computer Society.

[5] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In FatmaÖzcan, editor, *SIGMOD Conference*, pages 143–154. ACM, 2005.

[6] M. Bovee, R. P. Srivastava, and B. Mak. A conceptual framework and belief-function approach to assessing overall information quality. *International Journal of Intelligent Systems*, 18(1):51–74, jan 2003.

[7] R. Bruni and A. Sassano. Errors detection and correction in large scale data collecting. In *IDA*, volume 2189 of *Lecture Notes in Computer Science*, pages 84–94. Springer, 2001.

[8] A. Cleve, J. Henrard, and J.-L. Hainaut. Data reverse engineering using system dependency graphs. pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[9] A. Cleve, J. Henrard, D. Roland, and J.-L. Hainaut. Wrapper-based system evolution - application to codasyl to relational migration. In *Proc. of the 12th European Conference in Software Maintenance and Reengineering (CSMR'08)*, pages 13–22. IEEE Computer Society, 2008.

[10] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 315–326. ACM, 2007.

[11] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. An extensible framework for data cleaning. Technical Report RR-3742, INRIA, 1999.

[12] J.-L. Hainaut. Introduction to database reverse engineering. LIBD Publish., 2002.

[13] J.-L. Hainaut. The transformational approach to database engineering. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 95–143. Springer, 2006.

[14] J.-L. Hainaut, A. Cleve, J. Henrard, and J.-M. Hick. Migration of legacy information systems. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 105–138. Springer, 2008.

[15] J. Henrard and J.-L. Hainaut. Data dependency elicitation in database reverse engineering. In *Proc. of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*, pages 11–19, Washington, DC, USA, 2001. IEEE Computer Society.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[17] W. L. Low, M.-L. Lee, and T. W. Ling. A knowledge-based approach for duplicate elimination in data cleaning. *Inf. Syst.*, 26(8):585–606, dec 2001.

[18] D. Milano, M. Scannapieco, and T. Catarci. Using ontologies for xml data cleaning. In *OTM Workshops*, pages 562–571. Springer, 2005.

[19] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. Int'l Conf. Declarative and Object-oriented Databases*, 1995.

[20] R. J. Price and G. G. Shanks. Empirical refinement of a semiotic information quality framework. In *HICSS*. IEEE Computer Society, 2005.

[21] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[22] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 381–390. Morgan Kaufmann, 2001.

[23] T. C. Redman. *Data Quality for the Information Age*. Artech House, Inc., Norwood, MA, USA, 1997. Foreword By-A. Blanton Godfrey.

[24] M. Scannapieco, P. Missier, and C. Batini. Data quality at a glance. *Datenbank-Spektrum*, 14:6–14, aug 2005.

[25] P. Thiran, G.-J. Houben, J.-L. Hainaut, and D. Benslimane. Updating legacy databases through wrappers: Data consistency management. In *Proc. of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 58–67. IEEE Computer Society, 2004.

[26] Y. Wand and R. Y. Wang. Anchoring data quality dimensions in ontological foundations. *Commun. ACM*, 39(11):86–95, nov 1996.

[27] R. Y. Wang and D. M. Strong. Beyond accuracy: What data quality means to data consumers. *Journal of Management Information Systems*, 12(4):5–30, 1996.