2013 IEEE International Conference on Software Maintenance

Understanding Schema Evolution as a Basis for Database Reengineering

Maxime Gobert, Jérôme Maes and Anthony Cleve
PReCISE Research Center
University of Namur
Namur, Belgium
Email:{gobertm,maesj,acl}@info.fundp.ac.be

Jens Weber
Department of Computer Science
University of Victoria
Victoria, BC, Canada
Email: jens@uvic.ca

***Abstract*—Software repositories can provide valuable information for facilitating software reengineering efforts. In recent years, many researchers have started to follow a holistic approach, considering diverse software artifacts and the links existing between them. However, when analyzing data-intensive systems, comparatively little attention has been devoted to the analysis of an important system artifact: the database. Even fewer approaches attempt to uncover facts about the evolution history of database schemas. We have developed a tool-supported method for analyzing and visualizing database schema history. This paper reports early results of applying and validating this method. We discuss our experiences to date and point out several novel research perspectives in this domain.**

***Index Terms*—data reengineering, mining software repositories, schema evolution**

I. Introduction

Understanding the evolution history of a complex software system can significantly aid and inform maintenance and reengineering. Software repositories such as configuration management systems and issue trackers provide opportunities for historical analyses of system evolution. Most research work in this area has concentrated on program code, design and architecture. Fewer studies have focussed on database systems and schemas. This is an unfortunate gap as databases are often at the heart of today's information systems.

We have developed a tool-supported method for analyzing the evolution history of database schemas in order to improve program comprehension and inform software reengineering efforts. This paper describes our method, its implementation, and its application to a real-world case study.

II. Approach

Historical data about the evolution of database schemas can be helpful in software comprehension, but is rarely considered in current methods. For example, database schemas of long-term projects often contain "deprecated structures" that have been semantically superseded by other structures. Nevertheless, the deprecated structures are kept in order to maintain old data. Identification and documentation of such deprecated structures is important for program comprehension and reengineering. Other uses of historical schema analysis may provide insight into questions such as why some parts of the schema are apparently missing important constraints (such as foreign key declarations), while other parts comprise such constraints. (Usually, older parts of database schemas are less constrained as database management systems at that time may not have been able to compile and enforce such constraints.)

These two concerns are merely examples on how historical schema analysis may provide important clues in information system reengineering projects. The overall question for our research is thus: *How can we extract, represent and exploit the history of a database schema?*

Our general approach consists of extracting and comparing the successive versions of the database schema from the versioning system, in order to produce the so-called *global historical schema*. The latter is a visual and browsable representation of the database schema evolution over time. It contains all database schema objects (e.g., tables, columns and constraints) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime, which in turn serve as a basis for the visualization of the schema and its further analysis. This historical schema can be queried in order to derive valuable information about the evolution of the database, potentially raising other interesting system-specific questions to investigate.

The global process that we follow to build the historical database schema of a system consists of several steps:

1) *SQL code Extraction & Cleaning*: We first extract all the SQL files corresponding to each system version, by exploiting the versioning system used. Those files may then need to be slightly edited, in case the SQL syntax used does not match with the SQL parser considered[1].
2) *Schema Extraction*: We extract the logical schema corresponding to each SQL file obtained so far.
3) *Schema comparison*: We compare the successive logical schemas while incrementally building the resulting historical schema.
4) *Visualization & Exploitation*: The historical schema can then be visualized and further analyzed, depending of the project-specific needs.

Each historical schema object (e.g., table, column) is annotated with several meta-attributes:

[1]We use the schema extractor of DB-MAIN (http://www.db-main.eu)

- *NbVersions*: the total number of versions of the schema where the object can be found.
- *creationSchema*: references the first (oldest) schema version where the object appears.
- *creationDate*: the date the oldest schema version where the object appears (the date of creationSchema).
- *lastAppearanceSchema*: the last (most recent) schema where the object appears.
- *lastAppearanceDate*: the date of the *lastAppearanceSchema*.
- *severalLives*: true when the object has existed, has been removed, before being restored (false otherwise.)
- *isOpen*: is a helper attribute needed for the algorithm, meaning that the object has already a value in *lastAppearanceSchema* and *creationSchema*.

Algorithm 1 Deriving the global historical schema from n successive schema versions.

Notations.

- Let S_i be a database schema version, defined as a set of schema objects (including a set of tables and their respective columns).
- Let $date(S_i)$ be the release date of schema version S_i.

```
Require: $S_1, S_2, \ldots S_n$: $n$ successive schema versions.
Ensure: $S_H$: the corresponding global historical schema.
    // initializing $S_H$
 1: $S_H \leftarrow S_n$
 2: for all $o \in S_H$ do
 3:     $lastAppearanceSchema(o) \leftarrow S_n$
 4:     $lastAppearanceDate(o) \leftarrow date(S_n)$
 5:     $nbVersions(o) \leftarrow 1$
 6:     $isOpen(o) \leftarrow true$
 7: end for
    // iterating from the last version to the initial version
 8: for all $i \in \{n-1 \ldots 1\}$ do
 9:     // objects o appearing in more than one version
10:     for all $o \in S_i \cap S_H$ do
11:         $nbVersions(o) \leftarrow nbVersions(o) + 1$
12:         if $isOpen(o) == false$ then
13:             $severalLives(o) \leftarrow true$
14:             $isOpen(o) \leftarrow true$
15:         end if
16:     end for
17:     // objects o deleted before the last version
18:     for all $o \in S_i \setminus S_H$ do
19:         $S_H \leftarrow S_H \cup o$
20:         $lastAppearanceSchema(o) \leftarrow S_i$
21:         $lastAppearanceDate(o) \leftarrow date(S_i)$
22:         $nbVersions(o) \leftarrow 1$
23:     end for
24:     // objects o created after the initial version
25:     for all $o \in S_H \setminus S_i$ do
26:         if $isOpen(o) == true$ then
27:             $creationSchema(o) \leftarrow S_{i+1}$
28:             $creationDate(o) \leftarrow date(S_{i+1})$
29:             $isOpen(o) \leftarrow false$
30:         end if
31:     end for
32: end for
33: // Final step
34: for all $o \in S_H$ do
35:     if $isOpen(o) == true$ then
36:         $creationSchema(o) \leftarrow S_1$
37:         $creationDate(o) \leftarrow date(S_1)$
38:     end if
39: end for
```

Algorithm 1 formalizes our procedure for deriving a historical database schema S_H from n successive schema versions. This derivation algorithm is based on a pairwise comparison of all those schema versions in reverse chronological order.

The *initialization* step of the algorithm (lines 1-7) consists of considering the most recent schema S_n (augmented with its meta-attributes) as the initial historical schema S_H. We then iterate on all the previous schemas in reverse chronological order (lines 8-32), while comparing the current schema S_i with the current historical schema S_H. The comparison is made by iterating on each schema object of both schemas. Several situations may occur for a given schema object o:

1) o belongs to S_i and belongs to S_H (i.e., $o \in S_i \cap S_H$). In this case, o has appeared in more than one schema version. For such an object, we increment the *nbVersion* meta attribute. If its meta-attribute *isOpen* is $false$, this means that o has *several lives*: it had been removed after version i before reappearing later on, and version i corresponds to the end of (one of) its *previous life*. Therefore, we set its *severalLives* meta-attribute to true.
2) o belongs to S_i but does not belong to S_H (i.e., $o \in S_i \setminus S_H$). In this case, o has been deleted after version i, and it is the first time we encounter it. We then add o to the global historical schema together with its initial meta-attribute values.
3) o belongs to S_H but does not belongs to S_i (i.e., $o \in S_H \setminus S_i$). In this case, we can derive that o has been created in version $i+1$. We set its *isOpen* meta-attribute to $false$, in order to be able to identify its previous lives, if any.

The *Final* step is performed once all schema versions have been compared to the current historical schema. This step considers all schema objects of the historical schema for which the *isOpen* meta-attribute is still $true$. All those objects have actually been created in the initial schema version S_1. One therefore needs to initialize their *creationSchema* and *creationDate* meta-attributes accordingly.

III. Application and validation

We implemented our algorithm as a Java plugin to DB-MAIN and applied it to a case study of significant complexity: the OSCAR system [13]. OSCAR is an Electronic Medical Record (EMR) system for primary care clinics. One problem we faced was a lack of documentation and the sheer size of the database schema (over 465 tables and many thousands of columns). Moreover, at the time of conducting our study, the database schema of OSCAR contained little information on relationships between tables (foreign keys) and no documentation was available.

We analysed the history of the OSCAR database schema during a period of 9 years and 3 months (21/08/2003-21/11/2012). During this period, a total of 532 different schema versions can be found in the project's GitHub repository. However, we decided to consider only one version per month in our historical analysis. We therefore analysed 112 successive schema versions. Among these 112 versions, 8 schema versions proved to be identical to their previous version. The earliest schema version analyzed (21/08/2003) includes 88 tables, while the latest schema version considered (21/11/2012) comprises 460 tables.

Once the historical schema was derived, we applied a procedure that colourizes each historical schema object, depending on its age and its liveness. Fig. 1 shows a colourized version of the OSCAR historical schema, as derived by our tool, and that can be browsed and queried using DB-MAIN. All schema objects depicted in green constitute the tables and columns that are present in the latest schema version. All red schema objects have been deleted. The colour shade corresponds to the age of the objects. A dark red schema object is a table or column that has been deleted a long time ago. A light red object is an object that has recently been removed from the schema. An object depicted in green corresponds to a column or a table that is still present in the latest schema version. The darker the green, the older the corresponding table or column is, and vice versa. A schema object coloured in orange is a deleted object that had several lives.

The resulting historical schema was effective in helping us to answer important semantic questions about the database schema. For example, we found multiple seemingly unrelated schema structures covering the same semantic issue in the database. In one case, one schema structure revolved around tables entitled "*immunizations*" and "*configimmunization*" while another schema structure revolved around tables entitled "*preventions*" and "*preventionsext*". As immunizations are essentially disease preventions, we felt that there should be a relationship between these structures, or that one structure may in fact be deprecated and superseded by another. The historical schema allowed us to conclude the latter and identify the deprecated structure.

We also provide the user with a historical schema querying tool, allowing the extraction of interesting statistics regarding the evolution of the schema of interest. Some of those statistics are given below. Fig. 2 (left) provides an aggregate information about the creation and deletion of tables. One can easily notice that OSCAR tables are rarely removed. The evolution of the schema consists (most of the time) of adding tables, while not replacing or splitting them up. The total number of deleted tables is around 30, and we can again quickly identify the major release time periods. The observation is similar for the ratio of created and deleted columns, as shown in Fig. 2, right. The number of column creations is, indeed, often greater than the number of column deletions. During the last 10 releases, however, the removal of columns becomes more intensive: 867 columns were deleted between schema version 106 and schema version 107.

IV. Related work

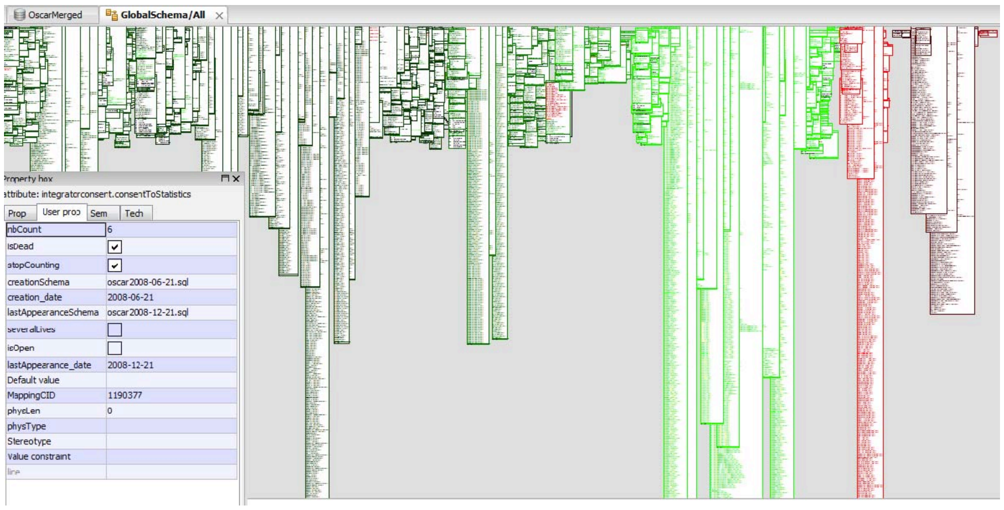
Research on software evolution has become popular thanks to Lehman's laws of software evolution [9]. This has inspired many researchers to validate these laws on open source software systems [7], [5], [14]. Today, it has given rise to an empirical research branch on software repository mining, investigating a wide variety of topics such as test and production code co-evolution [16], code cloning analysis [6], bug prediction techniques [4], change-proneness and fault-proneness [8], and many more.

While the literature on database schema evolution is very large [12], few authors have systematically *observed* how developers cope with database evolution issues in practice. Existing work in this domain [2], [10] analyzed the evolution of rather *small* database schemas. Curino *et. al* [2] present a study of the structural evolution of the Wikipedia database, with the aim to extract both a micro-classification and a macro-classification of schema changes. The authors propose, in addition to a schema evolution statistics extractor, a tool that operates on the differences between subsequent schema versions and semi-automatically extracts the set of possible schema operations that have been applied. For instance, simultaneously deleting an existing column and creating a new column may actually translate the renaming of the existing column. In this study, a period of 4 years has been considered, corresponding to 171 successive versions of the Wikipedia database schema. The latter is rather limited in size: from 17 to 34 tables depending on the schema version considered. The total number of columns in the schema does not exceed 250, whatever the version. Lin *et. al* [10] study the co-evolution of database schemas and application programs in two open-source applications, namely Mozilla and Monotone. The number of schema changes reported is very limited. In Mozilla, 20 table creations and 4 table deletions are reported in a period of 4 years. During 6 years of Monotone history, only 9 tables were created while 8 tables were deleted.

In this paper, we introduce the concept of global historical schema, and propose a scalable tool allowing to derive such a schema from the versioning system, in a browsable and queriable format. The concept of the global historical schema and its coloured visualization is analogous to the approach of visualizing program change sets at the architectural level with UML class diagrams [11], and is inspired from a myriad of other software visualization tools supporting the understanding of large-scale software systems evolution, including CodeCity [15], the Evolution Radar [3] and ExtraVis [1].

V. Conclusions

Analyzing the evolution history of database schemas aids understanding of the current schema structure and informs maintenance and reengineering efforts. The historical schema is useful in answering questions about schema structures, for example as in the case of the "preventions" table structure replacing the "immunizations" structure (see OSCAR case study above), while the aggregate (macroanalysis) allows us to infer general trends, such as the tendency of not deleting schema structures, even if they are superseded. To some degree, the creation of new schema structures to semantically replace existing ones without removing them is similar to the well-known process of "cloning" in program code. Therefore, the OSCAR preventions tables and the immunization tables could be considered *database clones*. However, subtle differences



attribute: integratorconsert.consentToStatistics

nbCount	6
isDead	✓
stopCounting	✓
creationSchema	oscar2008-06-21.sql
creation_date	2008-06-21
lastAppearanceSchema	oscar2008-12-21.sql
severalLives	☐
isOpen	☐
lastAppearance_date	2008-12-21
Default value	
MappingCID	1190377
physLen	0
physType	
Stereotype	
Value constraint	
line	

Fig. 1. The OSCAR historical schema, as it can be viewed in DB-MAIN with the property box containing the meta attributes

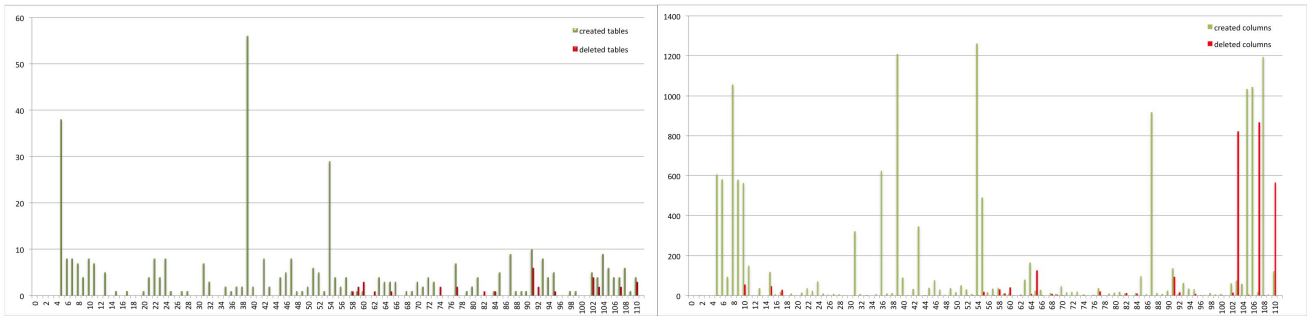


Fig. 2. Intensity of creation and deletion of OSCAR tables (left) and columns (right) over time.

exist to the concept of program clones. For example, program clones are usually still made up of functional code (as opposed to dead code), while the superseded database clone may indeed be considered "dead schema" from the point of view of at least a newer installation of the information system software, i.e., an installation that does not have to deal with legacy data that uses the superseded database structure. Nevertheless, our method may be a first step towards analysing database schema history for the purpose of schema clone detection.

References

[1] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *TSE*, 37(3):341–355, 2011.

[2] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In J. Cordeiro and J. Filipe, editors, *ICEIS (1)*, pages 323–332, 2008.

[3] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *TSE*, 35(5):720–735, 2009.

[4] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.

[5] J. Fernández-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical studies of open source evolution. In *Software Evolution*, pages 263–288. Springer, 2008.

[6] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of ICSE '11*, pages 311–320, New York, NY, USA, 2011. ACM.

[7] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. ICSM'00*, ICSM '00, pages 131–. IEEE CS, 2000.

[8] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[9] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, Sept. 1984.

[10] D.-Y. Lin and I. Neamtiu. Collateral evolution of applications and databases. In *Proc. of IWPSE-EVOL'09*, pages 31–40. ACM, 2009.

[11] A. McNair, D. M. German, and J. Weber-Jahnke. Visualizing software architecture evolution using change-sets. In *Proc. of WCRE'07*, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.

[12] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, Dec. 2006.

[13] J. Ruttan. *The Architecture of Open Source Applications, Volume II*, chapter OSCAR. Lulu.com, 2012.

[14] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi. Assessing architectural evolution: a case study. *ESE*, 16(5):623–666, Oct. 2011.

[15] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *ICSE*, pages 551–560. ACM, 2011.

[16] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *ESE*, 16(3):325–364, 2011.