sql-schema-comparer: Support of Multi-Language Refactoring with Relational Databases

Hagen Schink
Institute of Technical and Business Information Systems
Otto-von-Guericke-University
Magdeburg, Germany
hagen.schink@gmail.com

***Abstract*—Refactoring is a method to change a source-code's structure without modifying its semantics and was first introduced for object-oriented code. Since then refactorings were defined for relational databases too. But database refactorings must be treated differently because a database schema's structure defines semantics used by other applications to access the data in the schema. Thus, many database refactorings may break interaction with other applications if not treated appropriately. We discuss problems of database refactoring in regard to Java code and present *sql-schema-comparer*, a library to detect refactorings of database schemes. The sql-schema-comparer library is our first step to more advanced tools supporting developers in their database refactoring efforts.**

I. Introduction

Relational databases provide sophisticated functions for data persistence and retrieval [1]. In relational databases, data is stored according to a relational schema [2]. A relational schema predefines the logical structure of data.

Software applications are subject to change because of changing requirements and business conditions. However, adapting software applications may also require the adaption of the relational schema and vice versa.

Refactoring is a methodology to transform source code in such a way that the source-code's functionality is preserved [3]. Refactorings are specific code transformations and were first defined for object-oriented source code. But refactorings exist for relational databases too [4], [5]. Relational refactorings enable developers and database administrators to improve the structure of a relational schema without changing the informational semantics of the relational schema. Thus, a relational schema provides the same information before and after a relational refactoring.

Different APIs for many programming language exist to access relational databases. However, how APIs access data in the relational database depends on the structure defined by the relational schema. Thus, if the relational schema changes, e.g. because of refactoring, database access may be broken. Broken database access may only be noticed by integration tests or at runtime.

State-of-the-art IDEs provide information about syntactical changes or errors before integration or runtime tests take effect. Hence, the information can enable software developers to recognize and correct syntactic and some semantic errors already during development. We argue that information about syntactical changes in database schemes can ease the problems of database refactoring.

In the following we first present two techniques for accessing a relational database with the programming language Java. Then, we describe how database refactoring affects database access from Java. In Sec. III we give a more general explanation of problems resulting from database refactorings. Furthermore, we propose an approach to automatically detect database schema modifications affecting database interaction and present the *sql-schema-comparer* library that implements tool support for our approach. In Sec. IV we present related work before we conclude our work and present our future plans for the sql-schema-comparer library in Sec. V and Sec. VI.

II. Database Interaction and Refactoring

In this section we explain how data is retrieved from a relational database using the programming language Java. We also explain how changes of the relational schema affect the interaction between Java code and databases.

A. Examples of Database Interaction in Java

We consider two approaches to database interaction with Java: (1) a direct approach utilizing SQL statements and (2) an abstracted approach utilizing an object-relational mapper. The following examples are based upon two representatives of these approaches: (1) the Java Database Connectivity (JDBC) and (2) the Java Persistence API (JPA).

1) Direct Database Interaction with JDBC: With JDBC developers define database requests with SQL. In JDBC a database query returns an object of type `ResultSet` representing the SQL query result. Listing 1 presents a possible definition of a query that retrieves all first names of employees starting with the letter *M*.

Listing 1: JDBC: Parameterized query for first names

```
1  String stmt = "SELECT firstname FROM employees "
2                  + "WHERE firstname LIKE ?";
3  PreparedStatement query = con.prepareStatement(stmt);
4
5  query.setString(1, "M%");
6  ResultSet result = query.executeQuery();
```

Listing 2: Annotated class `Department`

```
1  @Entity
2  @Table(name="departments")
3  public class Department implements Serializable {
4
5    /* Snip */
6
7    private int id;
8    private String name;
9
10   public void setId(int id) { this.id = id; }
11
12   @Id
13   public int getId() { return id; }
14
15   public void setName(String name) { this.name = name; }
16
17   public String getName() { return name; }
18 }
```

Because of missing abstraction, JDBC requires detailed knowledge about the relational schema. For instance, in Listing 1 Line 5, the passed parameter must be an instance of class `String` because column `firstname` stores characters. Thus, a software developer must enforce type correctness.

2) Abstracted Database Interaction with JPA: JPA provides means to define a mapping between relational tables and object-oriented classes. This mapping is called *object-relational mapping* (ORM). Thus, in contrast to JDBC, developers receive objects instead of plain data from a relational database.

Listing 2 shows an implementation of a class representing a department. In JPA, each class annotated with `@Entity` becomes part of the ORM, thus, class properties are mapped upon columns in the mapped relational table. Hence, regarding Listing 2, we expect a table `departments` (Line 2) with the two columns `id` (Line 10 and 13) and `name` (Line 15 and 17)[1].

B. Database Refactoring and Interaction

The Database Refactoring website lists 69 database refactorings.[2] In the following we give examples on how database refactorings affect database interaction. To keep the following discussion concise, we focus only on two database refactorings, namely the Split Column Refactoring and the Introduce Column Constraint Refactoring.

1) Split Column Refactoring: The purpose of the Split Column Refactoring is to separate information that is combined in one table column. Let's assume we have a database column `amount` storing an amount of money and its currency. We want to split the column into two columns `amount` and `currency` to make queries for the currency easier. Existing JDBC queries like in Listing 3 need to be adapted because the column `amount` does not hold any currency information after refactoring. A possible adaption is given in Listing 4.

With JPA, we need to extend the ORM to map to the new column `currency`. Therefor, we need a new getter/setter pair, e.g. `getCurrency`/`setCurrency`.

Listing 3: Query accounts by currency

```
SELECT amount FROM accounts
WHERE amount LIKE '%EUR';
```

Listing 4: Modified query for accounts by currency

```
SELECT currency FROM accounts
WHERE currency = 'EUR';
```

2) Introduce Column Constraint Refactoring: The purpose of the Introduce Column Constraint Refactoring is to define a new column constraint to increase data quality. Let's assume we have a database table `accounts` with a column named `type`. The column `type` distinguishes different kinds of accounts by an uppercase letter. To ensure that only valid types are stored in table `account`, we can apply a column constraint like in Listing 5.

Listing 5: *CHECK* constraint on column `type`

```
'type' text CHECK(type = 'A' OR type = 'B' OR type = 'C')
```

A column constraint helps users with direct database access to preserve data quality. But applications accessing a table with constraints may not know about the restrictions enforced by the constraints. Such applications and their users are not able to handle the column constraints adequately. Developers of such applications may have to adapt the user interface and the applications database access to allow only such account types introduced by the Introduce Column Constraint Refactoring. Thus, the introduction of a column constraint by a refactoring may result in considerable effort to adapt related applications independent of the kind of database interaction.

III. Semantics-Preservation in Databases

In Sec. II-B, we showed that a database refactoring may result in considerable effort to adapt interacting software applications. We argue that this is the result of different meanings of semantics-preservation for software applications and database

[1]Because method `getId()` (Line 13) is annotated with `@Id`, JPA uses property access. Thus, columns are defined by getter/setter pairs.

[2]http://databaserefactoring.com/, 03/23/2013

schemes. This difference of semantics-preservation originates from a different semantics idea that is expressed in the term *informational semantics* [4].

The informational semantics of a database is preserved if the same information is available before and after a database transformation. We call a database transformation preserving the informational semantics a database refactoring. For instance, the Split Column Refactoring (c.f. Sec. II-B1) preserves the informational semantics because after the refactoring the information from the original column is just distributed over two columns. The Introduce Column Constraint Refactoring (c.f. Sec. II-B2) preserves the informational semantics because the refactoring just states explicitly an implicit assumption about a column's content.

However, although database refactorings preserve a database's informational semantics, application developers still may need to adapt the application code to the schema modifications. The question how the application code needs to be adapted may only be answered by the application developers themselves. Thus, application developers need a tool that makes schema modifications visible in such a way that application developers can decide whether and how to adapt the application code to the schema modification. In the following, we discuss which schema modifications affect the interaction with databases and, thus, need to be considered by application developers. Then, we present `sql-schema-comparer`, a library to compare databases schemes, that provides functions to detect database modifications.

A. Schema Modifications Affecting Database Interaction

In Sec. II-B we discussed the Split Column and Introduce Column Constraint Refactoring. In the following, we discuss general database refactoring categories in contrast to specific database refactorings.

Database refactorings can be distinguished by five categories: *Architectural*, *Data Quality*, *Performance*, *Referential Integrity*, and *Structural* [4]. To keep the following discussion concise, we consider the categories Data Quality and Structural.

In simple terms, with Data Quality Refactorings we set constraints on a column's content and with Structural Refactorings we modify a tables structure. Thus, Data Quality and Structural Refactorings may affect interacting applications differently.

1) Data Quality Refactorings: Data Quality Refactorings modify column data and column constraints. Except for the Consolidate Key Strategy Refactoring no columns are created or removed. Thus, if one of the following conditions is fulfilled database queries still work in an application:

1) a column constraint is removed,
2) application provides data conforming with constraints,
3) modified data formats can be processed by the application.

The first and the second condition are related: An application needs to comply with column constraints or it will not be able to insert/update data after refactoring. The first case is trivial because any data of the column's type will comply to non-existing constraints. The third condition points to issues after data format alignment: If the data format is changed, an application may not be able to process a new format, although the application may be able to insert/update data. For instance, assume we normalize telephone numbers stored in a database. After normalization applications may not be able to process the new format, although unnormalized data can still be entered into the database.

Data Quality Refactorings may not affect database interaction on first sight because refactorings may only affect certain corner cases. Therefore, after a Data Quality Refactoring, developers have to check the validity of all interactions.

2) Structural Refactorings: Structural Refactorings modify the structure of single tables as well as the whole database schema. Apart from the Introduce Calculated Column and Introduce Surrogate Key Refactoring, all Structural Refactorings rename, move, or drop schema elements, i.e. columns or tables. Therefore, after a structural refactoring an application will not be able to reference the modified schema element. Thus, Structural Refactorings affect database interaction if applied on a schema element that is accessed by an application.

B. SQL Schema Comparison

In Sec. III-A we discussed the effects of Data Quality and Structural Refactoring in regard to database interaction. In the following, we discuss important features a software tool has to provide for minimal support of developers adapting the interacting application code.

Effects of Data Quality Refactoring are not visible on first sight, thus, developers have to make qualified decisions about the impact of a Data Quality Refactoring. Therefor, tool support needs to find and present modified column constraints. Tool support for Structural Refactoring should be able to detect missing schema elements and to reveal the new position of moved schema elements. Therefor, a tool needs to be able to reveal the connection to other tables (foreign key relations) to show information about the new schema element's position.

Changes due to refactoring may be revealed by database schema comparison, thus, comparing the initial and the refactored database schema. In general, techniques for schema comparison are summarized under the term *schema matching* [6].

We state that adaption of database interaction after database refactoring is a manual task that is not likely to be completely automated in all cases. Because the adaption is a manual task, gathering information for adapting database interaction should be automated to provide a benefit. Different methods exist to support the comparison of two database schemes [6]. But these methods may depend on user interaction to complete a schema match, because these methods assume arbitrary schema modifications. But a database refactoring is a well defined schema modification that can consist of the following steps:

1) Create, Drop, Rename Table
2) Create, Drop, Rename, Move Column
3) Add, Drop, Change Column Constraint

A software tool can recognize single steps automatically if it has access to each single schema state after a step has been applied.

C. The sql-schema-comparer Library

We present *sql-schema-comparer*, a software library to compare and check SQL schemes.[3] The library allows to compare:

1) two SQL schemes,
2) an SQL statement and an SQL schema

with each other. The library provides support for SQLite statements and schemes and elementary support for JPA annotations in Java files.

The schema comparison and statement validation is performed on a graph representation. The schema comparison algorithm is able to automatically detect small schema modifications as described in Sec. III-B.

We argue that the sql-schema-comparer library provides the necessary functionality to build the basis for more sophisticated tools that, additionally, provide visual hints to inform developers about SQL schema refactorings.

1) Implementation Details: The library translates schema and statement information into a graph representation using the *jgrapht* graph library.[4] The graph contains nodes of type `ISqlElement`[5] implemented in the library and edges of type `DefaultEdge` (jgrapht). Tab. I lists all graph elements and their implementation in the sql-schema-comparer library.

Database	Graph	sql-schema-comparer
Table	Node	`SqlTableVertex`
Column	Node	`SqlColumnVertex`
Table-column relation	Edge	`TableHasColumnEdge`
Foreign-key relation	Edge	`ForeignKeyRelationEdge`

Table I: Graph elements and their respective Java types.

Nodes can represent tables or columns. Column nodes contain a list of column constraint information. Edges can represent a table-column relationship or a foreign-key relationship. Foreign-key relationships contain a relation between the referencing column to the referenced table and a reference to the foreign-key column in the referenced table. Fig. 1 shows an example graph that contains two table nodes *t1* and *t2* and four column nodes *c1*, *c12*, *c21*, and *c2*. The columns *c1* and *c12* are part of table *t1* and the columns *c21* and *c2* are part of table *t2*. Additionally, the graph contains an edge representing a foreign-key relationship between column *c12* and table *t2* by foreign-key column *c21*.

The sql-schema-comparer library is able to create schema graphs of SQLite databases, SQL SELECT statements, and JPA entity source files (Java classes annotated with `@Entity` from JPAs namespace `javax.persistence`). Tab. II lists all front-end implementations available in the sql-schema-comparer library. Fig. 2 shows the schema graph creation process. Note that, optionally, we are able to pass a database schema graph to the SQL Statement front-end. The SQL Statement front-end uses the database schema graph to augment the SQL statement's schema graph with type information that are not present in an SQL SELECT statement.

[3]https://github.com/hschink/sql-schema-comparer

[4]http://jgrapht.org

[5]We skip the full namespace of sql-schema- comparer's internal types and mark classes of the jgrapht graph library accordingly.

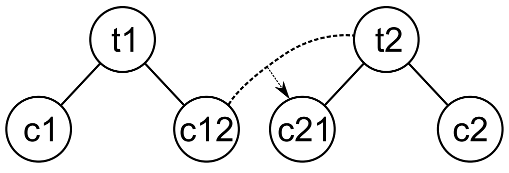

Figure 1: Database schema with a Foreign Key Relationship.

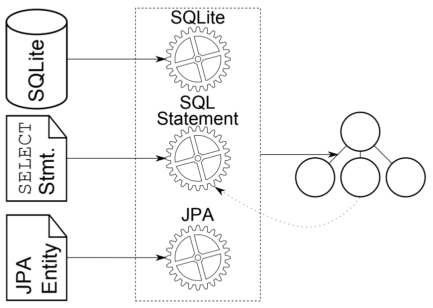

Figure 2: Schema graph creation process.

Source Type	sql-schema-comparern Front-end
SQLite	`SqliteSchemaFrontend`
`SELECT` statement	`SqlStatementFrontend`
JPA entity	`JPASchemaFrontend`

Table II: Available front-ends for schema graph creation.

The SQL schema comparison algorithm compares two SQL schemes by comparing each node of one schema graph with the nodes in the other schema graph respecting the table-column and foreign-key relationship. Taking into account our assumption, after refactoring we find a mismatch for at most one node. For each matching column node the algorithm compares the list of column constraints to detect constraint modifications.

The SQL statement validation compares an SQL statement with an SQL schema. The validation tries to match each node of the statement graph with a node of the schema graph. Additionally, if the algorithm cannot find a matching column node (a node with the same identifier and table node), the algorithm tries to find the column in referenced tables. Therefor, the algorithm checks for each column that matches the missing columns identifier if it is reachable via a foreign-key relationship. Fig. 3 shows an example schema graph with column node *c12* moved from table node *t1* to table node *t2*. The algorithm first looks for all column nodes matching the identifier *c12*. In respect to Fig. 3, the algorithm finds two possible match candidates in table *t2* and *t3*. Next, the algorithm checks if a foreign-key relationship exists. Because only table node *t2* is related to table node *t1* by a foreign-key relationship, column node *c12* of table node *t2* is identified as moved column.

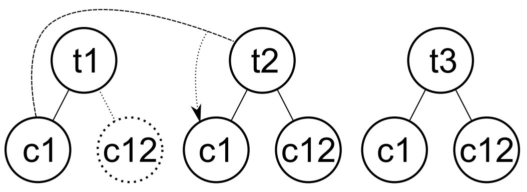

Figure 3: Resolving moved nodes by foreign-key relationships.

According to the classification by Rahm et al. [6], the SQL schema comparison algorithm and SQL statement validation possess the following properties. Matching granularity is different for tables and columns: Table nodes of on schema are only compared with table nodes of the other schema, thus, table comparison does not involve column comparison. In contrast, the comparison of column nodes also involves table node comparison to detect moved columns. Thus, the algorithms combine *element-level matching* (matching only table nodes with each other) as well as *structure-level matching*. In addition to the node's type, matching involves the node's identifier, i.e., table and column name. Because the matching algorithm combines element-level and structure-level matching and name matching, the algorithm can be called a *hybrid matcher*. Finally, because we assume that only a single node changed between two schemes, we restrict matching cardinality to 1:1.

2) Use Cases and Examples: The sql-schema-comparer library allows to compare two SQL schemes or an SQL statement with an SQL schema. In the following we describe the libraries application within a third-party application. For first tests you may use the library from the command-line.[6] Additionally, unit tests describe possible uses cases and expected results.[7]

a) Comparing SQL Schemes: For schema comparison we need to create a graph representation for each schema. Schema graphs are instances of class `SimpleGraph` (jgrapht) with the node type `ISqlElement` and edge type `DefaultEdge` (jgrapht). We may create instances manually or by parsing SQLite files. The latter is described in the following.

First of all we need to create an instance of the SQLite front-end as described in Listing 6, Line 1. The method `createSqlSchema()` of the front-end returns the schema instance for the SQLite file (Listing 6, Line 2).

The schema comparison is implemented in class `SqlSchemaComparer` which takes two schema instances on construction (Listing 7, Line 1). The field `comparisonResult` of an `SqlSchemaComparer` instance contains the match result. The match result contains the affected tables, columns, and column constraints. Because a comparison can only detect one refactoring step, the result can only contain one table or column.

[6]For details on the command-line usage refer to https://github.com/hschink/sql-schema-comparer#commandline-usage.

[7]Unit tests can be found in the project's subdirectory `src/test`.

Listing 6: Create a schema instance of an SQLite file

```
1  ISqlSchemaFrontend frontend = new SqliteSchemaFrontend(
       DATABASE_FILE_PATH);
2  Graph<ISqlElement, DefaultEdge> schema = frontend.
       createSqlSchema();
```

Listing 7: Compare two schema instances

```
1  SqlSchemaComparer comparer = new SqlSchemaComparer(
       schema1, schema2);
2  SqlSchemaComparisonResult result = comparer.
       comparisonResult;
```

b) Comparing an SQL Statement with an SQL Schema: To compare an SQL Schema with an SQL statement we need to extract a schema instance for the database (see Listing 6) as well as for the SQL statement. Latter is possible with the `SqlStatementFrontend` front-end as shown in Listing 8, Line 1. The front-end takes at least an SQL statement. Additionally, it is possible to pass a database schema, so type and constraint information can be extracted for the columns mentioned in the SQL statement.

The database and statement comparison is implemented in class `SqlStatementExpectationValidator` (Listing 9, Line 1). The class takes a database schema instance on construction. Calling the method `computeGraphMatching()` with the SQL statement schema returns a matching result (Listing 9, Line 2). The result contains missing tables or columns and columns that appear to be moved to different tables.

IV. Related Work

Two fields of research are addressed by this paper: multi-language refactoring and schema matching. In the following, we discuss both fields separately.

A. Multi-Language Refactoring

The *Cross-Language Link* (XLL) framework allows multi-language refactoring if language or technology-specific refactorings are available [7]. It is not discussed how the XLL framework can support developers if a refactoring exists for a database schema but not for object-oriented code or vice versa. Furthermore, the XLL framework has no support for semi-refactored states, e.g., the database is refactored but the object-oriented code does not match with the refactored schema.

Many approaches exist that describe multi-language refactoring for programming languages of a common paradigm or with a common technological base. In [8]–[11] approaches are presented that use source-code meta-models, namely FAMIX, MOOSE, UML, and the Common Meta-Model to describe and modify code of object-oriented programming languages. The common idea of these approaches is to describe a refactoring once for the common meta-model and to re-use it on the

Listing 8: Create a schema instance of an SQL statement

```
ISqlSchemaFrontend frontend = new SqlStatementFrontend(
        STATEMENT, null);
Graph<ISqlElement, DefaultEdge> schema = frontend.
        createSqlSchema();
```

Listing 9: Compare a database and a statement schema

```
SqlStatementExpectationValidator validator = new
        SqlStatementExpectationValidator(dbSchema);
SqlStatementExpectationValidationResult result = validator.
        computeGraphMatching(statementSchema);
```

different concrete languages. Because of this design syntactical elements of languages of different paradigms cannot be described and, thus, not be refactored.

B. Schema Matching

An overview of schema matching approaches are given in [6], [12]. The presented schema matchers implement a general approach to schema matching. Thus, in contrast to the sql-schema-comparer library, they allow to compare arbitrary schemes with each other. Furthermore, because the presented schema matchers only determine match candidates, user interaction is still necessary to select valid match candidates. To the best of our knowledge no schema matching approach exists dedicated to the specifics of database refactoring.

V. Conclusion

Relational databases are an integral part of many modern software applications. Database interfaces exist for the majority of programming languages. However, modern IDEs provide no means for developers to recognize database refactorings or even simple schema modifications. Thus, developers are on their own to detect SQL schema modifications and to adapt their applications to a modified schema.

In this paper, we discussed the influence of database refactorings to interacting software applications. For this purpose we discussed two database refactorings in detail: (1) Split Column and (2) Introduce Column Constraint Refactoring. Furthermore, we discussed a minimal feature set that a tool should provide to support developers adapting their software application to a refactored database schema. Then, we concluded that we may only be able to automate schema modification detection under certain assumptions. Finally, we presented the *sql-schema-comparer* library that allows to detect schema modifications and to validate SQL statements against SQL schemes.

VI. Future Work

Using the *sql-schema-comparer* library requires a reasonable amount of user interaction and, thus, is not practically usable. In our ongoing work we focus on integrating the sql-schema-comparer library into the Eclipse IDE. The idea for Eclipse integration is to provide database refactoring support for Java for two use cases: (1) direct database access with the *Java Database Connectivity* (JDBC) and (2) abstract database access with the *Java Persistence API* (JPA).

With JDBC we use SQL statements (c.f. Sec. II-A1) and with JPA we use JPA entities (c.f. Sec. II-A2) to interact with databases. The sql-schema-comparer library is able to compare SQL statements and JPA entities with a given SQLite database. Thus, in a future Eclipse plug-in, we are able to not only highlight those SQL statements and JPA entities that do not match with a given database schema but to give more detailed information about the mismatch. In a first step we may let the developer select the SQL statement or JPA entity as well as the SQLite database file that need to be compared.

Assume that a developer has access to each single change of an SQLite file's schema.[8] With the sql-schema-comparer library we are able to create a change history of the database's schema. The change history may provide useful information for adapting mismatching SQL statements or JPA entities to the new database schema.

References

[1] E. F. Codd, "The 1981 ACM Turing Award Lecture Relational Database : A Practical Foundation for Productivity," vol. 25, no. 2, 1982.

[2] ——, "A relational model of data for large shared data banks. 1970." *M.D. computing : computers in medical practice*, vol. 15, no. 3, pp. 162–6, 1970. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/9617087

[3] M. Fowler, *Refactoring: Improving the Design of existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[5] S. Ambler and P. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[6] E. Rahm and P. a. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, Dec. 2001.

[7] P. Mayer and A. Schroeder, "Cross-Language Code Analysis and Refactoring," *SCAM*, pp. 94–103, 2012.

[8] S. Ducasse, M. Lanza, and S. Tichelaar, "MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," *CoSET*, 2000.

[9] S. Tichelaar, "Modeling Object-Oriented Software for Reverse Engineering and Refactoring," Ph.D. dissertation, University of Berne, Switzerland, 2001.

[10] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards Automating Source-Consistent UML Refactorings," *UML*, 2003.

[11] D. Strein, H. Kratz, and W. Lowe, "Cross-Language Program Analysis and Refactoring," *IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 207–216, 2006.

[12] P. Shvaiko and J. Euzenat, "A survey of schema-based matching approaches," *Journal on Data Semantics IV*, no. October 2004, 2005.

[8] With SQLite the current schema can be preserved by copying the file itself.