Documenting Database Usages and Schema Constraints in Database-Centric Applications

Mario Linares-Vásquez[1,2], Boyang Li[1], Christopher Vendome[1], Denys Poshyvanyk[1]
[1]The College of William and Mary, Williamsburg, VA, USA
[2]Universidad de los Andes, Bogotá, Colombia
{mlinarev, boyang, cvendome, denys}@cs.wm.edu

ABSTRACT

Database-centric applications (DCAs) usually rely on database operations over a large number of tables and attributes. Understanding how database tables and attributes are used to implement features in DCAs along with the constraints related to these usages is an important component of any DCA's maintenance. However, manually documenting database related operations and their asynchronously evolving constraints in constantly changing source code is a hard and time-consuming problem. In this paper, we present a novel approach, namely *DBScribe*, aimed at automatically generating always up-to-date natural language descriptions of database operations and schema constraints in source code methods. *DBScribe* statically analyzes the code and database schema to detect database usages and then propagates these usages and schema constraints through the call-chains implementing database-related features. Finally, each method in these call-chains is automatically documented based on the underlying database usages and constraints.

We evaluated *DBScribe* in a study with 52 participants analyzing generated documentation for database-related methods in five open-source DCAs. Additionally, we evaluated the descriptions generated by *DBScribe* on two commercial DCAs involving original developers. The results for the studies involving open-source and commercial DCAs demonstrate that generated descriptions are accurate and useful while understanding database usages and constraints, in particular during maintenance tasks.

CCS Concepts

•**Software and its engineering** → **Software maintenance tools; Documentation;**

Keywords

Database-centric applications, Documentation, SQL-data statements, Schema constraints

1. INTRODUCTION

Database-centric applications (DCAs) are software systems that rely on databases to query, persist, and modify records using database objects such as tables, columns, and constraints, among others. These database objects represent the underlying domain model of DCAs, including business rules and terms. Modern DCAs contain databases (DBs) comprised of a large number of tables and attributes, and DCA architectures are commonly used for different types of systems ranging from large enterprise systems to small mobile apps [11, 14, 25, 38]. As for the communication between DCAs and databases, DCAs use database connectivity APIs (e.g., JDBC,) and object-relational mapping (ORM) frameworks (e.g., Hibernate) to access and manipulate database objects from source code entities that (i) are spread across the entire system, or (ii) belong to a data access layer [13,40] when DCAs follow a multilayered architecture.

Previous work extensively studied the co-evolution of source code and DB schemas demonstrating that: (i) schemas evolve frequently, (ii) the co-evolution oftentimes happens asynchronously (i.e., code and schema evolve collaterally) [18, 24, 34, 49], and (iii) schema changes have significant impact on DCAs' code [18, 38, 49]. Therefore, co-evolution of code and DB schemas in DCAs often leads to two types of challenging scenarios for developers, where (i) changes to the DB schema need to be incorporated in the the source code, and (ii) maintenance of a DCA's code requires understanding of how the features are implemented by relying on DB operations and corresponding schema constraints. Both scenarios demand detailed and up-to-date knowledge of the DB schema; however, there is an inherent gap between the skills and activities performed by developers and DB administrators. While developers are usually more knowledgeable of DCAs' internals and logic, DB administrators, in turn, are more knowledgeable of the DB model and its evolution [49]. Hence, one critical artifact that developers need to be able to maintain DCAs effectively is up-to-date and complete documentation of the DB schema and any constraints. However, such documentation can be only obtained by navigating and comprehending a multitude of constantly evolving DB artifacts (e.g., data dictionaries, schema models), which is prohibitively tedious, error-prone and time-consuming task, in particular, for large-scale DBs.

Source code comments are another source of documentation that could help developers understand nuances of the model and DB usages. However, recent studies on the co-evolution of comments and code showed that the comments are rarely maintained or updated when the respective source code is changed [21, 22]. In addition, a previous study of 3.1K+ open source projects [36] demonstrated that 77% of source code methods invoking SQL-statements (e.g., `CREATE`, `DROP`, `INSERT`, `SELECT`, `UPDATE`) do not have header comments; moreover, existing comments, rarely get updated whenever related source code is modified. Despite the availability of DB schemas, two-thirds of 147 surveyed DCA developers indicated that tracing DB schema constraints (e.g., unique values, non-null keys, varchar lengths) along source code method call-chains was a "moderate" or a "very hard" challenge [36]. In addition, lack of usage of referential integrity in schemas impact the understanding of the schemas [18], which is a common issue in open-source DCAs. In our study of 3.1K+ open source DCAs (Section 2), we found that only 23.29% use primary keys and 12.17% use foreign keys in their schemas.

The main contribution of this paper is a novel approach, *DBScribe*, aimed at automatically generating always up-to-date documentation describing database-related operations and schema constraints that need to be fulfilled by developers during software maintenance tasks. *DBScribe* statically analyzes the source code and database schema of a given DCA in order to generate method level documentation of the SQL-statements related to methods' execution, and the schema constraints imposed over these statements. The documentation is generated for methods executing database operations locally as well as for methods invoking the operations via delegation, which supports developers maintaining different layers or modules of a DCA (e.g., data access or GUI). To the best of our knowledge, this is the first work that proposes a solution for automatically documenting source code methods related to database operations in a DCA and the corresponding schema constraints.

We validated *DBScribe*'s documentation in a study involving 52 participants, who were asked to analyze those in terms of completeness, expressiveness, and conciseness (Section 5) for five open source DCAs. In addition, we validated *DBScribe* by interviewing the original developers of two commercial web-based DCAs from a Colombian company (Section 5). The results show that descriptions generated by *DBScribe* are considered to be complete, concise, readable, and useful for understanding database usages and constraints for a given DCA, in most of the cases. Moreover, participants consider that this type of description is useful mostly for maintenance tasks such as program comprehension, debugging, and documentation.

2. WHY DO WE NEED TO DOCUMENT DB RELATED CODE?

In this section, we provide motivating examples from real-world DCAs to illustrate some of the common problems that developers face when maintaining DCAs. We also describe the studies that we performed to understand developers' preferences and practices while documenting DCAs.

2.1 Motivating Examples

Database operations in source code can be encapsulated in data access layers or are spread across the whole system when it lacks design patterns. In both cases, the high-level features (and the corresponding business logic) provided by the DCAs might use/persist/modify data in different database objects. Therefore, a high-level feature im-

```
public ReturnPK createReturn(Handle userHandle, Handle languageHandle, SchedulePK
schedulePK, String versionCode) [...] {
        [...]
         try{
           int versionId = getVersionId(con, versionCode);
            [...]
            if (returnId != -1 && hasVersion(con, returnId, versionId)) {
                   throw new FinaTypeException(Type.RETURNS_RETURN_NOT_UNIQUE);
            }
            ps = con.prepareStatement("select id from IN_RETURNS
                   where scheduleID=?");
            ps.setInt(1, schedulePK.getId());
           [...]
             if (returnId != -1) {
                pk = new ReturnPK(returnId);
            } else {
                ps = con.prepareStatement("select max(id) from IN_RETURNS");
                [...]
                PreparedStatement insert = con.prepareStatement("insert into
                         IN_RETURNS (id,scheduleId,versionId) values(?,?,?)");
                 [...]
            }
            createDefaultValues(con, pk.getId(), versionId);
            changeReturnStatus(userHandle, languageHandle, pk,
                     ReturnConstants.STATUS_CREATED, " ", versionCode);
            updateReturnVersions(con, String.valueOf(returnId));
        } catch (RuntimeException e) {
            log.error(e.getMessage(), e);
            if (pk != null)
                deleteReturn(pk, versionCode);
            throw e;
        } catch (FinaTypeException e) {
            log.error(e.getMessage(), e);
            if (pk != null)
                deleteReturn(pk, versionCode);
            throw e;
        }
          [...]
    }
```

Figure 1: FINA's method `fina2.returns.ReturnSessionBean.createReturn`

plementation relies on different calls to database operations that represent a call tree; the root of the call tree is a method (often in the GUI) that triggers/starts the feature, and methods with direct calls to SQL-statements or ORM API methods are nodes in the tree at different levels. In addition to the combination of database operations, high-level feature implementations depends on the constraints imposed by the data model. In that sense, maintaining DCAs' source code requires understanding (i) the database operations that are related to a feature, (ii) the source code methods exposing the database operations at different layers in the system architecture, (iii) the constraints imposed by the database schema, (iv) the expected database constraints that are not defined in the schema (e.g., when the schema does not use foreign keys), and (v) the source code location where the constraints should be handled or accounted for.

Moreover, inferring database schema constraints and understanding the database operations involved in a DCA features is a challenging task, since less information about the DB can be inferred from source code methods at higher levels of the call-chains. We illustrate this case on two methods from two open-source DCAs: FINA 3.4.6 [2] and Xinco rev.700 [10].

FINA is a system for Bank Supervision Authorities that is used to collect data from banks and make decisions based on the reports generated from the collected data. FINA has been in active development since 2002, and the core system uses a J2EE two layer (i.e., GUI and business) architecture based on Enterprise JavaBeans 3.1. The schema has neither referential integrity nor uniqueness constraints, and it is composed of 52 tables and 261 attributes. For illustration purposes, we analyzed manually the code of the `ReturnSessionBean.createReturn` method; the code is depicted in Figure 1. String literals for SQL-statements are highlighted in red. Apparently, only three database operations are performed on the same table `IN_RETURNS`. However, we found that the execution of the method may trigger 20 database-related operations (2 `UPDATE`, 12 `SELECT`, 3 `INSERT`, 3 `DELETE`) on eight different tables when considering delegated executions (see methods in bold in Figure 1). Ten

```java
public void deleteFromDB(boolean delete_this, XincoDBManager DBM,int userID)
                throws XincoException {
        int i=0;
        try {
            Statement stmt;
            fillXincoCoreNodes(DBM);
            fillXincoCoreData(DBM);
            for (i=0;i<getXinco_core_nodes().size();i++) {
                ((XincoCoreNodeServer)getXinco_core_nodes().elementAt(i))
                .deleteFromDB(true, DBM,userID);
            }
            for (i=0;i<getXinco_core_data().size();i++) {
                XincoIndexer.removeXincoCoreData([...]);
                XincoCoreDataServer.removeFromDB([...]);
                [...]
            }
            if (delete_this) {
                XincoCoreAuditServer audit= new XincoCoreAuditServer();
                stmt = DBM.con.createStatement();
                stmt.executeUpdate("DELETE FROM xinco_core_ace WHERE
                                    xinco_core_node_id=" + getId());
                stmt.close();
                audit.updateAuditTrail("xinco_core_node",new String [] {"id ="+getId()},
                        DBM,"audit.general.delete",userID);
                stmt = DBM.con.createStatement();
                stmt.executeUpdate("DELETE FROM xinco_core_node WHERE id=" + getId());
                stmt.close();
            }
            DBM.con.commit();
        } catch (Exception e) {
            [...]
        }
    }
```

Figure 2: Xinco's method `com.bluecubs.xinco.core.server.XincoCoreNodeServer.deleteFromDB`

of those operations are performed in one-level chain (i.e., *a* calls *b*, and the SQL-statement is in *b*), six of those operations are implemented via a two-level chain call (i.e., *a* calls *b*,*b* calls *c*, and the SQL-statement is in *c*), and one operation is done in a three-level chain. According to the schema, the only constraint is that the attribute `NOTE` in the table `IN_RETURN_STATUSES` should not exceed 4,096 characters.

Another example is the `XincoCoreNodeServer.deleteFromDB` method from Xinco rev.700 [9]. Xinco is a document management system, which has been in active development since 2004. The schema is composed of 23 tables and 135 attributes, and it uses referential integrity. The code of the method is depicted in Figure 2. A simple check reveals that the method invokes deletion of rows on tables `xinco_core_ace`, and `xinco_core_node`. However, five more tables are involved in SQL-statements via delegated calls (in bold). In total, 14 SQL-statements might be invoked when this method is called (12 statements via delegation). Six of those 14 statements are `DELETE`, four are `SELECT`, two are `INSERT`, and two are `UPDATE`. Deletion of rows in tables `xinco_core_ace`, and `xinco_core_node` are also performed via a call to `XincoCoreDataServer.removeFromDB`. Surprisingly, the analyzed method (`XincoCoreNodeServer.deleteFromDB`), which has a signature describing a deletion operation, also includes insertions and updates. Concerning the level of the delegated executions, there are seven call-chains with level 2, and four with level 3. In addition, there are 24 different constraints that should be taken into account: nine potential violations of referential integrity when inserting into `xinco_core_ace` and `xinco_core_node`, 12 attributes that should not be null, four potential violations of varchar limits, and two attributes that should be unique.

These examples illustrate that the process of understanding schema constraints and the SQL-statements associated with each method is a challenging, time-consuming, and error-prone task. While the schema of Xinco was useful for identifying the constraints, FINA's schema was not helpful at all. In the latter, the relationships between tables are managed by soft links, which makes understanding the code and schema harder; however, having up-to-date documentation at method level describing the database operations that are triggered (locally and via delegation) can reduce the time involved in navigating the code, and help developers to (i) understand methods' purpose, and (ii) identify implicit relationships and constraints in the database schema. The examples also corroborate that code search and navigation are necessary tasks for identifying relevant elements and the context during software maintenance tasks [23,28,37]. In the case of database-related code, maintenance tasks also require understanding of the constraints and details behind feature implementations involving DB operations, which adds even more complexity to the task.

2.2 Studying Developers' Preferences

In order to understand developers' preferences and practices while documenting database usages and constraints in source code, Linares-Vásquez *et al.* [36] surveyed developers of open-source DCAs. The main conclusions of the study were that (i) documenting methods with database accesses is not a common practice; and (ii) developers prefer to have documentation of schema constraints in the schema itself or external documentation, rather than in the source code. In addition, despite the high confidence of surveyed developers on database documentation, around 66% of respondents answered that tracing schema constraints along call-chains in the source code (which is a common task faced by DCA maintainers) is a moderate/very hard challenge.

We conducted a survey and asked 147 developers of open-source DCAs the following questions: *Do you (would you) find method documentation, which help understanding the database schema constraints and functional usage of a method, beneficial for software evolution and maintenance tasks? and Why?*, where 85 (57.82%) participants responded "yes", and 62 participants answered "no". Table 1 summarizes the rationale behind these responses. We analyzed the textual answers using qualitative coding; it is worth noting that in some cases the rationale included more than one reason (i.e., more than one code). For the cases in which we were not able to identify the rationale because the answers were not clear, we coded the rationale as "Unclear".

The answers describe different perspectives of DCA developers with a slight tendency towards accepting the idea of methods' documentation describing database usages and schema constraints. Their preferences are directly related to the strategies commonly used to maintain DCAs: (i) changes are first done in the schema and then the source code is adapted to schema changes (bottom-up); (ii) the source code is modified first and then the schema is modified (top-down); (iii) source code is modified without changing the schema, but the code maintenance requires understanding how the features are implemented using database operations and which constraints are involved.

According to the positive answers, documenting database usages and constraints at method level is considered to be a good practice, and is particularly useful for program comprehension and maintenance. Some participants stated that:

"If the method accessing a certain part of the database isn't properly commented, it may take quite a bit of time to go through each step of the code to figure out what it is doing. If the method and how it is accessing the database was properly documented, life for maintenance and general development would be much easier."

"Yes, because they help facilitate any communication required with the DB developers/admins, and help make updating the code less of a headache."

Also, this type of documentation can be useful for under-

Table 1: **Summary of developer survey answers**

Answer	Rationale
Yes (85)	Program comprehension (21), unclear (18), no rationale (14), good practice (11), source code maintenance (8), database model understanding (7), database evolution impact analysis (4), documentation (3), errors prevention (3), schema evolution understanding (1), debugging (1)
No (62)	Not useful (9), unclear (8), preference for self-documented code (8), schema is enough (7), ORM (7), concerns separation (7), external documentation (6), collateral evolution and outdated comments (6), others (4)

standing the data schema and the impact of schema changes in the code: *"Methods do different things than databases. We need documentation in both places."*; *"Would help in noting the restrictions if the database scheme is not available."*

Negative answers show "more radical" positions toward the preference for self-documented code, lack of confidence in the usefulness of this type of documentation, and a clear separation between schema responsibilities and source code. Seven answers point that the schema is enough for understanding database operations and constraints; seven answers assert that annotations from ORM are sufficient; and seven answers explicitly mention that database usages and constraints should be documented externally. The following answer is a representative example of a negative position by some developers: *"Code is a more reliable source of truth. Comments are not. If I would need such info I will trace or walk from public API till DB layer"*. Another interesting reason, which was provided by the respondents is the one noting that collateral evolution may lead to outdated comments in source code: *"Code is no place to document the database schema. Tiny changes to the schema could make all comments untrue."*; *"The comments could be out of date; check the database for the truth."* These answers (positive and negative) support the need for an automated approach for generating up-to-date and accurate documentation for database usages and constraints at method level.

2.3 Referential Integrity in DCAs

We also investigated the frequency of DCA schemas with referential integrity, motivated by a previous study describing an experience on integrating software for a primary care research network with the OSCAR EMR system, where the lack of relationships and documentation in the OSCAR DB was an initial impediment to understand the schema [18]. Therefore, we mined the latest snapshots of 3,113 DCAs looking for database schema files declaring primary and foreign keys. It is worth noting that all the 3,113 DCAs use JDBC; therefore, the schema is not created automatically in those DCAs, conversely to DCAs that delegate schema creation to ORM frameworks.

We found a total of 5,086 SQL files declaring explicit primary key constraints and 2,196 SQL files declaring explicit foreign key constraints in these 3,113 projects. The projects contain multiple files representing the schema or multiple dumps for several database engines or different schema snapshots; however, the SQL files define primary keys for only 725 projects (23.29% of the projects), and foreign keys declarations were found only in 379 projects (12.17% of the projects). Therefore, the lack of referential integrity is common in open-source DCAs, which might be a potential problem for developers maintaining those DCAs similarly to the case presented by Cleve *et al.* [18].

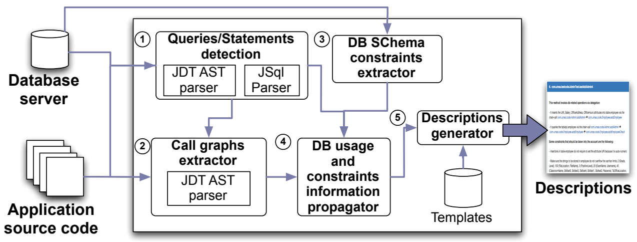

Figure 3: *DBScribe* **components and workflow.**

3. DBSCRIBE: DOCUMENTING DATABASE USAGES AND SCHEMA CONSTRAINTS

DBScribe provides developers with updated documentation describing database-related operations and the schema constraints imposed on those operations. The documentation is contextualized for specific source code methods; in other words, the documentation is generated considering the local context of the methods and the operations delegated through inter-procedural calls and the subsequent call-chains that involve at least one SQL-statement. Therefore, our method-level documentation can provide developers with descriptions that work at different layers for a given DCA. Concerning the usefulness, we designed *DBScribe* to help developers when (i) understanding how features are implemented using SQL operations, and (ii) understanding schema constraints that need to be satisfied in both specific methods of the source code and all the operations involved. Also, *DBScribe* is suitable for on-demand execution by developers that require up-to-date documentation. For instance, Table 3 lists the execution time in seconds of *DBScribe* when running on a MacBookPro laptop with a 2.4GHz Intel Core 2 Duo processor and 4GB of DDR3 RAM. While it took two authors of this paper 10 minutes on average to analyze the examples in Figures 1 and 2, *DBScribe* was able to analyze and generate complete documentation for these two DCAs in 130.78 and 31.41 seconds, respectively.

The architecture of *DBScribe* is depicted in Figure 3. *DBScribe*'s workflow is composed of five phases: ① SQL-statements and the methods executing them are detected in the source code statically; ② a partial call graph with the call-chains including the methods executing SQL-statements (locally and by delegation) are extracted from the source code statically; ③ database schema constraints are extracted by querying the master schema of the database engine that has an instance of the database supporting the DCA under analysis; ④ the constraints and SQL-statements are propagated through the partial call graph from the bottom of the paths to the root; and ⑤ the local and propagated constraints and SQL-statements (at method-level) are used to generate natural language based descriptions. Current *DBScribe*'s implementation covers SQL-statements invoked by means of JDBC and Hibernate API calls. Future work will support other ORM frameworks such as JPA and iBATIS.

Each phase in *DBScribe*'s workflow is described in the following subsections; however, we first provide **formal definitions** that are required to understand the proposed model:

- M is the set of all the source code methods/functions in a DCA, and m is a method in M;

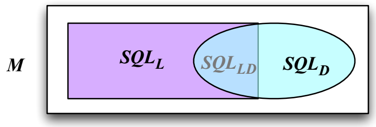

Figure 4: Sets of methods in a DCA. M **is the set of all the methods in the DCA,** SQL_L **is the set of methods executing at least one SQL-statement locally, and** SQL_D **is the set of methods executing at least one SQL-statement by means of delegation.**

- SQL_L is the set of methods $m \in M$ that execute at least one SQL-statement locally (Figure 4);
- SQL_D is the set of methods $m \in M$ that execute at least one SQL-statement by means of delegation through a path of the DCA call graph (Figure 4);
- G is the call graph of a DCA involving all the methods $m \in M$, and $G_{SQL} \subset G$ is the call graph including only the methods in $SQL_L \cup SQL_D$. It means that G_{SQL} is a (partial) call graph of all the methods in M that execute at least one SQL-statement locally or by delegation;
- P is the set of paths p (a.k.a., call-chains) starting at any method $m_i \in G_{SQL}$ and finishing at any method $m_j \in SQL_L$. It is possible that the number of methods in some p's is equals to 1 (i.e., $|p_i| = 1$); those cases represent unused methods, or methods in a upper layer of the DCA invoking SQL-statements. It is worth noting that P can be a disconnected graph;
- The methods in a path p are an ordered set defined by the binary relationship (represented with the symbol $<$) between a callee and a caller. For instance, given a method $m_i \in SQL_L$, m_j a caller of m_i, m_k a caller of m_j, and so on, until all the methods in the path p are exhausted, the ordered set is $m_i < m_j < m_k < ... < m_s$. The position in the ordered set, is the attribute l, which represents the "level" of the method in the path p, and is in the range $[0, |p| - 1]$. In the example, the l value for m_i is zero, and the l value for m_k is two. Conversely to the level, the depth d of a method in a path is the position in the ordered set but in the direction of callers or callees; for instance, the level l of m_i in our example is zero, but the depth d is $|p| - 1$.
- QS_m is the set of tuples $qs = \langle literal, T, A, type \rangle$ representing the SQL-statements executed locally in method m. Each tuple qs has a SQL string literal, tables (T) and attributes (A) from the database schema referenced in the literal, and the SQL-statement type.
- $\overrightarrow{C}_m$ is the set of methods called by method m (i.e., callees), and $\overleftarrow{C}_m$ is the set of methods calling method m (i.e., callers).

3.1 Detecting SQL-Statements

All the methods in M are analyzed by first identifying API calls that execute SQL-statements; using this approach, we avoid SQL-statements that are declared as strings but never executed. In addition to the API calls detection, for the case of Hibernate, *DBScribe* (i) analyzes configuration files and map POJO classes to database tables and attributes, and (ii) retrieves database constraints from XML mapping files or annotations defined in Java POJO classes.

In order to identify db-related API calls, we traverse the AST of each method in M; during the traversal, we keep a working map SV with all the `String` and `StringBuffer` variables instantiated in the method as well as the current values. We update the variables in SV after a new initialization or after a concatenation operation with the plus (+) operator or the `StringBuffer.append` method.

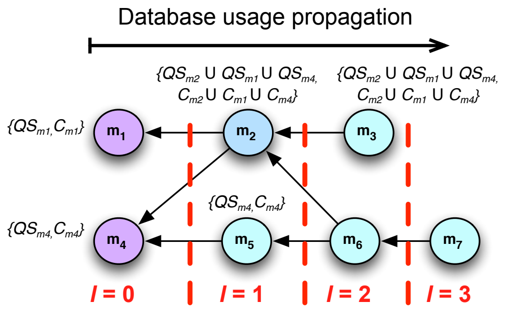
Database usage propagation
$\{QS_{m2} \cup QS_{m1} \cup QS_{m4}, C_{m2} \cup C_{m1} \cup C_{m4}\}$
$\{QS_{m2} \cup QS_{m1} \cup QS_{m4}, C_{m2} \cup C_{m1} \cup C_{m4}\}$
$\{QS_{m1}, C_{m1}\}$
m_1
m_2
m_3
$\{QS_{m4}, C_{m4}\}$
$\{QS_{m4}, C_{m4}\}$
m_4
m_5
m_6
m_7
$l = 0$
$l = 1$
$l = 2$
$l = 3$

Figure 5: Iterative propagation of database usage information over the ordered sets defined by the paths in the partial call graph

During the traversal, we also keep track of invocations to API methods that execute or prepare SQL-statements, e.g., `Statement.execute`, `Statement.executeQuery`, `Statement.executeUpdate`, and `Connection.prepareStatement`. If the String argument in the API call is a literal, we add the literal to the list of SQL-statements QS_m declared in the method m; if the string argument is an infix expression or a variable name, we infer the value of the argument by resolving the expression/variable state with the values in the map SV. The String literals and inferred variable values are used to build SQL literals, which are parsed by using the JSqlParser [3] to identify the statement type, and the tables and attributes involved in the SQL-statement (this information is required to generate the textual descriptions as described in Section 3.3). Then, we add the resolved SQL-statement (i.e., literal, tables, attributes, and type) to the list QS_m. One limitation in this procedure is that we do not perform inter-procedural analysis; thus, we do not resolve values that are returned by inter-procedural calls or values passed as arguments to the analyzed method (future work will be devoted to inferring the SQL queries/statements by using symbolic execution). This leads to cases in which some SQL literals are not parsed by JSqlParser. For instance, Table 3 lists in column "NP" those cases in which *DBScribe* was not able to parse the SQL-statements in seven DCAs that we analyzed, and column "S" lists the number of API calls that execute SQL-statements. More details about *DBScribe*'s evaluation are provided in Section 5.

The $\overrightarrow{C}_m$ and $\overleftarrow{C}_m$ sets for each $m \in M$ are also collected during the traversal to avoid a second pass on the DCA code. Both sets are used to generate the G_{SQL} graph required to propagate SQL-statements and constraints.

3.2 Propagating Constraints and SQL-Statements through the Call Graph

DB schemas contain a set of constraints that need to be fulfilled when realizing insertions, updates, or deletions on the DB. For instance, changing/deleting the value of a column that serves as a foreign key in other tables cannot be performed when there are references from other tables to that column. As mentioned in Section 2, this type of constraints cannot be inferred easily from the source code, and the constraints provide useful information that can help in understanding source code methods in abstract layers that

Table 2: Subset of Templates used by DBSCRIBE to generate the database-related descriptions at method level. Examples from the systems used in the study are also provided

Type	Template	Example
Section: Local SQL-statements		
Header	This method implements the following db-related operations:	This method implements the following db-related operations:
Insert	It inserts the ⟨attr⟩ attributes into table ⟨table⟩	It inserts the **Username**, **Passwd** attributes into table **logindetails**
Update	It updates the ⟨attr⟩ attribute(s) in table ⟨table⟩	It updates the **IsCurrent** attribute(s) in table **semester**
Section: Delegated SQL-statements		
Header	This method invokes db-related operations by means of delegation:	This method invokes db-related operations by means of delegation:
Query	It queries the table(s) ⟨table⟩ via ⟨method⟩	It queries the table(s) **People** via the call-chain JobApplication.addApplicationDetails → Student.checkIfStudent
Delete	It deletes rows from table(s) ⟨table⟩ via ⟨method⟩	It deletes rows from table(s) **gradingsystem** via a call to the GradeSystem.deleteGrade method
Section: Schema constraints		
Header	Some constraints that should be taken into the account are the following:	Some constraints that should be taken into the account are the following:
Varchar	Make sure the strings to be stored in ⟨table⟩ do not overflow the varchar limits: ⟨limits⟩	Make sure the strings to be stored in **employee** do not overflow the varchar limits: 2 (Grade, Level), 100 (FileLocation, FileName)
Non-null	Make sure the values in ⟨table⟩.⟨attr⟩ are not null	Make sure the values in **employee.Salary** are not null

are not close to the DB (e.g., $\mathbf{m}_7$ in Figure 5). Therefore, in addition to linking source code methods to SQL-statements, we extract from the DB schema — by querying the master schema in the DB server— the constraints that are defined on the attributes and tables in the sets $QS_m, \forall m \in SQL_L$, i.e., the constraints that apply to all the attributes and tables involved in SQL-statements executed by the methods in the application. In particular, we extract the following constraints: (i) auto-numeric columns, (ii) non-null columns, (iii) foreign keys, (iv) varchar limits, and (v) columns that should contain unique values. Consequently, each method m in SQL_L has a list X_m of schema constraints that apply to the SQL-statements executed locally by m.

Both QS_m and X_m sets contain information for all of the methods in SQL_L; however, the methods in SQL_L are not the only methods that can benefit from documentation describing DB usages and schema constraints. Developers inspecting, using, or updating methods that execute SQL-statements by means of delegation (see SQL_D in Figure 4), similarly to the motivating examples in Sec. 2.1, could require documentation describing DB usages and constraints across all methods involved in the DB-related call-chains. Thus, we propagate the information in the QS_m and X_m lists to all the methods in SQL_D. Notably, methods in the intersection of SQL_L and SQL_D have at least one local execution of an SQL-statement and at least one by delegation.

To describe the propagation algorithm, we use Figure 5 as a reference. The figure depicts a partial call graph with nodes representing source code methods and directed edges going from a caller to a callee. The background color of the nodes (the colors are the same from Figure 4) represents the set to which each method belongs. For instance, light purple is for methods in SQL_L, light blue for methods in SQL_{LD}, and cyan for methods in SQL_D. Only the methods in SQL_L (including SQL_{LD}) execute SQL-statements locally.

The propagation is done iteratively by using the node level l (see definition at the beginning of Section 3) as the iteration index, until the maximum l in the call graph is reached. This iterative execution over l assures that the values from methods with a lower level l are computed before the methods with a level $l+1$. This step is required because one method can have more than one callee in the graph G_{SQL}. The nodes (i.e., methods) with $l = 0$ are methods belonging to SQL_L, and, the lists QS_m and X_m for those methods were computed previously. However, the lists of queries/statements and constraints that concern the methods with $l > 0$ are the unions of the local QS_m and X_m lists (if any) and the lists propagated from the callees. For example, the node $\mathbf{m}_2$ calls $\mathbf{m}_1$ and $\mathbf{m}_4$, and belongs to SQL_{LD}, which means that $\mathbf{m}_2$ executes at least one SQL-statement locally and at least two by means of delegation in $\mathbf{m}_1$ and $\mathbf{m}_4$. Therefore, the complete sets of SQL-statements and constraints that concern $\mathbf{m}_2$ are $\{QS_{m_2} \cup QS_{m_1} \cup QS_{m_4}\}$ and $\{X_{m_2} \cup X_{m_1} \cup X_{m_4}\}$, respectively. The case for the methods in cyan is different, because they do not execute SQL-statements locally; consequently, the set of SQL-statements that concerns a method m in SQL_D is the union of SQL-statements and constraints from its callees, and the set of constraints that concerns the method is the union of the constraints from its callees. For example, $\mathbf{m}_5$ does not execute SQL-statements locally. Hence, the SQL-statements and constraints that concerns $\mathbf{m}_5$ are the same from its single callee $\mathbf{m}_4$. Note that we do not perform branch analysis of source code in *DBScribe*; we catch as many branches as possible, likely overestimating the results. In the future, we will rely on static analysis techniques to improve the precision [16].

3.3 Generating Contextualized Natural Language Descriptions

The final phase in *DBScribe* generates contextualized natural language descriptions by using predefined templates. In general, a description for a method m in G_{SQL} consists of three parts: (i) a block (i.e., a header plus a list of sentences) describing SQL-statements executed locally, (ii) a block describing SQL-statements executed by means of delegation and the path in the call graph to the execution, and (iii) a block describing the constraints that should be taken into account as a result of the SQL-statements that are executed locally or by delegation. *DBScribe* only generates descriptions for methods that are related to database operations.

A subset of *DBScribe* templates is listed in Table 2, while the complete list is provided in our online appendix [1]. Each template has tokens identified with ⟨...⟩, which are replaced with values from the sets of SQL-statements and constraints that concern a method m. In the case of execution by delegation, the templates include the token ⟨method⟩; this token is replaced by a single method call (see template Delegation-Delete) or by a call-chain that goes over a path from m to the method, where the corresponding SQL-statement is executed. The sentences in the constraints paragraph are generated based on the SQL-statements concerning the method m. For instance, the sentences generated with constraints-related templates in Table 2 are only included if the method executes insertions/updates locally or by delegation. Our current implementation of *DBScribe* generates the descriptions as HTML pages with hyperlinks to the methods in the call-chains (as in the delegation-related sentences); this allows for easy browsing and navigation of the call-chains.

4. EMPIRICAL STUDY

We conducted a user study to evaluate the usefulness of *DBScribe* at generating descriptions for source code methods that execute SQL-statements locally, by means of delegation, and the combination of both. The *goal* of this study is to measure the quality of the descriptions generated by *DBScribe* as perceived by developers. As for the *context*, we used seven DCAs listed in Table 3. The first five systems are open-source DCAs hosted at GitHub and SourceForge. The last two DCAs are industrial web Java applications developed by a Colombian company (*LIMINAL ltda*). It is worth noting that LOC reported in Table 3 only include .java files; for instance, web side files like JSP, HTML and CSS were not included. Also the numbers in columns "ML","MD", and "MLD" are the ones reported by *DBScribe*.

Table 3: Systems' statistics: Lines Of Code, TaBles in the DB schema, # of JDBC API calls involving SQL-Statements, # of SQL statements that *DBScribe* was Not able to Parse, # of Methods declaring SQL-statements Locally (ML), via Delegation (MD), Locally + Delegation (MLD), execution Time in sec.

System	LOC	TB	S	NP	ML	MD	MLD	T
`UMAS` [8]	32K	122	211	4	125	431	67	29.53
`Riskit rev.96` [7]	12.7K	13	111	2	35	9	44	15.02
`FINA 3.4.2` [2]	139.5K	52	710	26	312	118	99	130.78
`Xinco rev.700` [9]	25.6K	23	76	15	26	22	21	31.41
`OpenEmm 6.0` [5]	102.4K	68	200	110	73	12	1	104.78
`System 1*`	73.2K	53	398	27	262	660	24	71.07
`System 2*`	28.4K	24	164	8	106	247	44	40.13

We selected subject systems with the following constraints in mind: (i) the systems should rely on JDBC/Hibernate and MySQL for the data access layer, since the current version of *DBScribe* was designed to detect SQL-statements from JDBC/Hibernate API calls and extract schema constraints from MySQL DBs, and (ii) the systems should pervasively use SQL-statements.

4.1 Research Questions

In the context of our study, we formulated the following five research questions (RQ):

RQ$_1$ *How complete are the database usage descriptions generated by DBScribe?*

RQ$_2$ *How concise are the database usage descriptions?*

RQ$_3$ *How expressive are the database usage descriptions?*

RQ$_4$ *How well can DBScribe help developers in understanding database related source code methods?*

RQ$_5$ *Would developers of DCAs use DBScribe descriptions?*

RQ$_1$ to **RQ$_3$** aim at measuring the quality of the descriptions as perceived by developers that have explored the source code and the database schema. **RQ$_4$** aims at identifying whether the descriptions are useful for developers and the software development tasks that can take advantage of this type of description. **RQ$_5$** is for exploring the potential usefulness of *DBScribe* for supporting DCAs and potential adoption by industrial DCA developers and maintainers.

4.2 Data Collection

We used descriptions generated by *DBScribe* for methods of the open-source DCAs in an open survey with students, faculty, and developers. We randomly selected six methods from each system (30 descriptions in total from five open-source DCAs); in particular, we selected two methods from the GUI layer that are at the root of method call-chains invoking SQL-statements, two methods that are leaves of the call-chains (i.e., declare SQL-statements, but do not delegate declaration/execution to other methods), and two methods in the middle of the call-chains. This selection was aimed at evaluating *DBScribe*'s descriptions at different layers of DCAs' architectures. Also, we limited the survey to six descriptions per system to make sure our survey could be completed in one hour to avoid an early survey drop-out. For the evaluation, we relied on the same framework previously used for assessing automatically generated documentation [19, 41, 46, 54]. Therefore, the descriptions were evaluated in terms of *completeness*, *conciseness*, and *expressiveness*. In addition, we sought to understand the preferences of the participants concerning *DBScribe*'s descriptions.

We designed and distributed the survey using the Qualtrics [6] tool. We asked participants to evaluate *DBScribe*'s descriptions by following a two-phase procedure. In the first phase, we asked developers to *manually* write a summary documenting the SQL-statements executed (locally and by means of delegation) as part of a call to a given source code method and the constraints that should be considered by developers when understanding that method. In particular, each developer was provided with the source code of the DCA, an entity-relationship diagram, and six source code methods to document; we also provided the SQL script to create the database schema as an optional artifact that can be used during the task. We decided to use only six methods per DCA, because writing each summary requires detailed inspection of the source code and the databases. This phase was designed to make sure that the participants understood the source code before evaluating *DBScribe*'s descriptions.

In the second phase, we asked participants to compare their own (manual) summaries to *DBScribe*'s descriptions. For each *DBScribe* description, the participants rated the three quality criteria (i.e., completeness, conciseness, expressiveness) with the options listed in Table 4. In addition, we asked them to provide a rationale for their choices; the evaluation criteria and the rationale provided the answers to **RQ$_1$** to **RQ$_3$**. For **RQ$_4$**, we included two questions regarding the usefulness of the descriptions in the survey. To measure the programming experience of the participants, we included background/demographic questions [20].

For the case of the industrial systems (i.e., **RQ$_5$**), two original developers of `System 1` and `System 2` from *LIMINAL ltda* [4] were interviewed. We provided them with a complete *DBScribe* report (i.e., an HTML page with descriptions for all the methods with hyperlinks) and asked to read the report and analyze the code. The report also organizes the methods in the three groups in Figure 4 to enable easier browsing. While all the reports for the open-source DCAs are provided in our online appendix, we were not allowed to publicize the reports for the industrial DCAs.

During the interview, in addition to the questions from the open survey, we asked the following: (i) *Only focusing on the content of the document without considering the way it has been presented, do you think all the database-related methods are listed in the document?*; (ii) *Is the document useful for understanding the database usages in the system?*, (iii) *How could we improve the document?; (iv) What kind of information would you like to include/remove?*

4.3 Threats to Validity

In order to reduce the threats to *internal* validity and maximize the reliability of the results of evaluation, we confirmed that the participants explored and understood the source code before evaluating the summaries generated by *DBScribe*. In terms of evaluation, we used a well-known framework that has been applied previously to evaluate the quality of natural language summaries of software artifacts. Also, in order to avoid any type of bias because of the expectations of the participants during the study, we informed the participants that they had to evaluate generated descriptions only after completing the phase in which they needed to write their own summaries. Concerning the threats to *external* validity, we do not assert that the results in the second study apply to the entire community of Java developers. However, the set of participants is diverse in terms of academic/industry experience, and 42.3 percent of the participants have more than five years of experience in Java. Although the study was done on only five open-source and two industrial DCAs, when designing the study we selected a diverse set of source code methods belonging to different layers of the systems' architecture, which means that we evaluated methods executing SQL queries/statements locally, through delegation, and a combination of both.

5. EXPERIMENTAL RESULTS

We obtained responses from 52 participants: 15 responses for both `UMAS` and `Riskit`, eight responses for `Xinco`, and seven responses for the other two open-source DCAs, `Openemm` and `Fina`. In terms of background, we had the following distribution: three undergraduates and thirty-five graduate students (M.S/Ph.D), two post-docs, seven developers/industry researchers, and five faculty members. Three participants participated in the study twice (voluntarily), i.e., they analyzed the descriptions for two different systems. 24 of our participants (46.1%) asserted that they had past experience in industry. Concerning the programming experience in Java, 22 of participants (42.3%) had at least five years of experience; the mean value is four years of experience.

We evaluated the quality of *DBScribe* generated descriptions by considering three attributes: *completeness*, *conciseness*, and *expressiveness*. For *completeness*, we aimed at assessing whether the descriptions cover all the important information (**RQ**$_1$). For *conciseness*, we aimed at evaluating whether the descriptions contain useless information (**RQ**$_2$). For *expressiveness*, we aimed at checking whether the descriptions are easy to understand (**RQ**$_3$). Since we asked participants to evaluate three attributes for six descriptions for each DCA, we had a total of 312 answers for each attribute (6×52). Table 4 reports both raw counts and percentages of answers provided by the participants; detailed results are also available in our online appendix [1].

Table 4: Distribution of the Participants' Responses

Criteria	Options	Responses
Completeness	•Does not miss any important info	205 (65.7%)
	•Misses some important info	91 (29.2%)
	•Misses most important info	16 (5.1%)
Conciseness	•Contains no redundant info	221 (70.8%)
	•Contains some redundant info	77 (24.7%)
	•Contains a lot of redundant info	14 (4.5%)
Expressiveness	• Is easy to read	241 (77.3%)
	•Is somewhat readable	60 (19.2%)
	•Is hard to read and understand	11 (3.5%)

RQ$_1$ **(Completeness):** The results show that 65.71% answers agreed that *DBScribe*'s descriptions do not miss any important information, while only 5.13% answers indicated the documents missed the most important information. In other words, our approach is able to generate DB-related descriptions for source code methods that cover all essential information in most of the cases (**RQ**$_1$). We also examined answers with the lowest ratings. One comment mentioned: *"The description does not make it clear that the time-slot is not always added to the table."* The reason for this comment is that we did not apply branch analysis when generating descriptions. However, this would not influence the completeness, since we chose to over-approximate in order to catch important information.

When comparing the summaries written by participants to *DBScribe*'s descriptions, we found that only 24 out of 312 human-written descriptions include specific information about the schema constraints. Most of the human-written descriptions (i.e., 5 out of 7 for `RiskIt`) detailing constraints correspond to the methods at the lowest level of the call-chains (i.e., the methods executing statements locally). These findings corroborate our hypothesis that the methods in the higher levels in the call-chains may be more difficult to understand with respect to relevant DB operations and the schema constraints (Section 2). Therefore, *DBScribe* descriptions are not only useful for methods that are architecturally close to the DB, but also for source code methods in layers that are more close to the end-user (e.g., GUI layer).

RQ$_2$ **(Conciseness):** 70.83% of the answers asserted that *DBScribe*'s descriptions do not contain redundant information and only 4.49% answers indicated that the descriptions contain a lot of redundant information.

Again, we examined the answers with the lowest ratings. Participants' comments included the following: *"This data feels too low level."* We closely checked our generated documents for those methods. Our observation is that *DBScribe*'s documentation sometimes contains unnecessary information for the task we assigned. The extra information is unavoidable because the documentation is produced without taking into account a particular task on which a developer may be working. In addition, since we provided call-chains in the documentation, the descriptions for methods in the top level of the call hierarchy may appear rather verbose. For example, most of low ratings (10 out of 14) were for the descriptions of methods situated in the middle or higher levels of call-chains.

RQ$_3$ **(Expressiveness):** in 77.24% of the answers, *DBScribe*'s descriptions were evaluated as easy to read, while only 3.53% answers indicated that the descriptions were hard to read. We analyzed the user feedback from the participants who provided the lowest ratings for expressiveness. Those participants who thought some descriptions were hard to read claimed that the descriptions had a lot of information. Similar to *Conciseness*, our descriptions are attempting to capture more important information, which may come at the expense of *expressiveness*. We also observed that there were only two out of 11 responses with the lowest rating for the methods situated in the lowest level of call-chains over all systems. Thus, descriptions for methods only invoking SQL-statements locally are the easiest to read.

The following comments illustrate some of the reasons why participants evaluated *DBScribe*'s descriptions mostly positively (completeness, conciseness, and expressiveness):

"It is useful when we need to know all the entities (i.e. tables, constraints, indexes, etc) involved in a database operation. This information helps a lot if someone needs to modify/extend the code."

"This description definitely outperformed the description that I just made. The details shown by this description really help to understand all the entities involved in the creation of a new user, and this information is helpful when someone tries to modify/extend the source code."

"It summarizes the database accesses efficiently that might be spread over many different methods. Even if all related database operations are contained in the respective method directly, the summary is much easier to read than finding the specific statements in the code."

"The generated summaries are useful because they show all related db operations and also show another information related with the constraints, data types. With the constraints and validations allows to developer to understand any business logic restrictions."

RQ$_4$ (User preferences): 48 participants (92.3%) claimed that *DBScribe* generated descriptions would be useful for understanding the database usages in source code methods. When looking into the details for each system, we found that 100% of the answers were positive for `UMAS`, `Xinco`, and `Openemm`; we got three negative responses for `RiskIt` and one negative response for `Fina`. Although we do not have enough evidence to claim a relationship between users preferences and the type of DCA system for which descriptions are generated, the results suggest that *DBScribe* is more helpful specifically for DCAs with larger DBs and more complex chain-calls. In our case, `Riskit` has less complicated call hierarchy and database design than others (see Table 3).

We also asked the participants for which software engineering tasks they would use these descriptions. We categorized the answers in Table 5. The most answered tasks are related to incremental change, such as program comprehension, implementing new features, and impact analysis. The second most reported category is "Bugs", where participants mentioned debugging six times, and bug fixing four times. Examples of the answers are:

"This information can be useful when:-I need add a new feature, I can understand the related db actions of any existing method.-To fix a bug. To build a business process."

"Understanding the code in general. I could also imagine that it is particularly helpful for debugging database-related errors (wrong updates, wrong implications drawn from the data) as well as performance problems due to unnecessary database queries."

Table 5: Answers to *"What software engineering tasks will you use this type of summary for?"*

Category	Subcategories
Incremental change (21)	Program comprehension (11), Add new features (4), Impact analysis (4), Concept location (1), Change database schema (1)
Bugs (10)	Debugging (6), Bug fixing (4)
Maintenance (10)	Maintenance (4), Refactoring (2), Re-modularization (2), Re-engineering (2)
Others (15)	Documentation (9), Change db-related code (3), Test cases design (2), Systems integration (1)

"Bug fixing (to find out useful information possibly related with the bug) and refactoring/remodularization (in order to make sure that modification will not invalidate some constraints)"

RQ$_5$ (Usefulness and adoption for maintenance of real DCAs): Two practitioners from *LIMINAL ltda* analyzed the reports generated by *DBScribe* for two industrial DCAs. Both systems were developed using a multi-tier web architecture and MySQL as the database engine. The database schemas use referential integrity. `System 1` uses JSP, CSS and Javascript for the presentation tier; a set of servlets operates as controllers between the web components and a tier of business entities that implement the logic and persistence operations. `System 2` uses Java Server Faces for the presentation tier; JSF Beans and a JSF Front Controller are used as controllers between the GUI and application services (i.e., business tier); the application services invoke database operations by means of a persistence tier implemented with JDBC Data Access Objects; Value Objects are used to transfer data over all the tiers. `System 1` has been in production for about ten years, and `System 2` has been in production for over seven years. Both systems are currently maintained by *LIMINAL ltda*.

Concerning the practitioners, their current positions are project managers, but they were the original developers of both systems. They have ten years of industrial experience developing Java web applications. One of the practitioners asked us to anonymize his name; thus, we refer to him as Practitioner 1. The second practitioner is Néstor Romero, who also was the developer in charge of `System 2`'s maintenance for one year. We asked both practitioners to indicate the architectural layer(s) in which they are more proficient: Practitioner 1 responded "Business Layer, Data Access, and Utilities", Néstor responded "GUI and Business Layer".

Both practitioners ranked completeness, conciseness, and expressiveness for the two reports, giving them the highest values (i.e., does not miss any important info, contains no redundant info, is easy to read). Also, both agreed positively on the usefulness of the reports. For instance, Néstor noted *"Based on the descriptions you can be aware all dependencies a table could have. It would let you estimate in a better way the impact due to future changes."*; *"The text provided helps to create a basic understanding of the functionality"*. With respect to the software engineering tasks, they would use the reports for program understanding, impact analysis, and technical documentation. In particular, they mentioned: *"It helps you create a quick vision of the system with the basic method and code structure without looking at actual source code"*; *"Creating a data - dictionary for a system"*.

Finally, both practitioners agreed on features for improving navigation features in the reports. For instance, Participant 1 claimed *"It would be useful to search a table name in* *order to see what dependencies it has"*, and Néstor claimed *"The link system for call-chains works only in one way, one could get lost navigating a complex system as there is no visual or textual reference of my location within the entire document. A navigation tree might be useful in this case."* Practitioner 1 augmented the response with some desirable features: *"you should extend the approach to include JPA"*, and *"it would be better to have it in the IDE, something like right click->generate"*.

Summary of the results: *The DBScribe's descriptions are complete, concise and readable, in most of the cases. The participants consider the descriptions to be useful for understanding the DB usages in DCAs. Moreover, this type of descriptions is useful for understanding DB related source code. Concerning software engineering tasks, the participants consider that the summaries can be mostly useful for incremental change-related tasks, debugging, and bug fixing.*

6. RELATED WORK

Previous work related to *DBScribe* can be summarized in three different clusters: studies on analyzing co-evolution of DCA schemas and source code, automatic inference of relationships from database schemas, and automatic documentation/summarization of software artifacts. In the following, we describe those works and their relationship to *DBScribe*.

6.1 Co-evolution of Schema and Code

Recent studies showed a presence of strong evolutionary coupling between database schemas and source code [24,38, 49,53]. Maule *et al.* used program slicing and dataflow-based analysis to identify the impact of database schema changes [38]. Qiu *et al.* conducted an empirical study into co-evolution between DB schemas and source code demonstrating that DB schemas frequently evolve with many different types of changes at play [49]. Sjøberg mainly focused on database schema changes and presented a technique for measuring the changes of database schemas; a study on health management systems over several years showed additions and deletions to be the most frequent operations [53]. Cleve *et al.* presented a method to analyze the change history of DBs [18]. *These studies and findings serve as our main motivation for developing* DBScribe *to help developers understand evolving DB usages and schema constraints.*

6.2 Inferring Database Relations

The approach by Petit *et al.* first extracts DB table names and attributes from the DB schema; then, it builds semantic relationships between the entities by investigating set and join operations [48]. Alhajj *et al.* designed an algorithm to identify candidate and foreign keys of all relationships from a legacy DB [12]. Another group of studies focused on analyzing data in DBs and extracting associative constraints [11, 31]. The associative rule mining problem was first introduced by Agrawal *et al.*, where the associative rule mining algorithm is able to generate a set of implications $A \rightarrow B$ based on a given relational table [11]. Au *et al.* applied a fuzzy association rule mining technique to a bank DCA and identified some hidden patterns in the data [31]. Li *et al.* used association rule mining to correct semantic errors in generated data [32]. *These approaches represent an extension opportunity for DBScribe, since the relationships between DB objects can be inferred even when schemas lack referential integrity. We will extend DBScribe to automatically infer such relationships as part of the future work.*

6.3 Documenting Software Artifacts

Buse and Weimer proposed an approach for generating human-readable documentation of exceptions in Java [16]. More specifically, they used a method call graph and symbolic execution techniques to extract conditions of exceptions [16]. Then, they used predefined templates for generating natural language comments. Sridhara *et al.* [55], and McBurney and McMillan [39] designed approaches for automatically generating method comment summaries. Moreno *et al.* later extended the scope of the comment generation to class level granularity [41,43]. Rastkar *et al.* proposed summarization techniques for describing crosscutting concerns [50, 51]. Other artifacts have also been in focus of summarization/documentation techniques such as code fragments [57,58], bug reports [52], unit test cases [27,33,46], and developer discussions [45,56].

Differently from the previous work, a number of papers focused on documenting differences between program versions [15,17,19,26,35,42,44,47]. Linares-Vásquez *et al.* implemented a tool for automatically generating commit messages [19, 35]. Moreno *et al.* introduced an approach for automatic generation of release notes of Java systems [42]. Jackson and Ladd built a tool SematicDiff, which uses program analysis techniques for summarizing semantic differences between two versions of a system [26]. Kim *et al.* proposed and approach to infer structural differences and describe the changes by using logic rules [29, 30]. Buse and Weimer proposed another approach for generating documentation for program differences based on symbolic execution [15]. *However, none of the existing approaches focus on generating DB-related descriptions; DBScribe is the first to analyze source code and DB schemas for generating method-level documentation to support DCA maintenance.*

7. CONCLUSION

We presented *DBScribe*, a novel approach for automatically generating natural language documentation at source code method level that describe database usages and constraints for a given DCA. The descriptions are generated by detecting methods executing local SQL-statements and then propagating the schema constraints through all the methods that execute the SQL-statements by means of delegation. To evaluate *DBScribe*, we conducted a study involving 52 participants in which we asked them (i) to rate the completeness, conciseness, and expressiveness of the summaries, and (ii) to describe the usefulness of the summaries. We also asked developers from a Colombian company to evaluate the documentation we generated for two of their DCAs. The results show that *DBScribe* descriptions are useful for understanding database usages and constraints across a given system, and the descriptions can be used for supporting maintenance of DCAs' code. *DBScribe* provides a solution to a challenging and common problem by relying on static analysis of source code and database schemas, and automatic documentations techniques. *DBScribe* currently supports systems using the JDBC and Hibernate APIs; therefore, future work will support SQL-statements executed with other ORM frameworks and database engines. We will improve the SQL-statement detection by resolving SQL literals in source code that are declared with values returned by interprocedural calls or passed as arguments to methods.

8. REFERENCES

[1] Dbscribe online appendix. http://www.cs.wm.edu/semeru/data/ISSTA16-DBScribe.

[2] Fina http://sourceforge.net/projects/fina/.

[3] Jsqlparser. http://jsqlparser.sourceforge.net/.

[4] Liminal ltda http://www.liminal-it.com/.

[5] Openemm e-mail & marketing automation http://sourceforge.net/projects/openemm/files/OpenEMM%20software/OpenEMM%206.0/.

[6] Qualtrics. http://www.qualtrics.com.

[7] Risk it repository. https://riskitinsurance.svn.sourceforge.net.

[8] Umas repository. https://github.com/University-Management-And-Scheduling.

[9] Xinco rev 700 http://sourceforge.net/p/xinco/code/700/tree/trunk/.

[10] Xinco http://sourceforge.net/projects/xinco/.

[11] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[12] R. Alhajj. Extracting the extended entity-relationship model from a legacy relational database. *Information Systems*, 28(6):597–618, 2003.

[13] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2013.

[14] K. Bakshi. Considerations for big data: Architecture and approach. In *Aerospace Conference, 2012 IEEE*, pages 1–7. IEEE, 2012.

[15] R. Buse and W. Weimer. Automatically documenting program changes. In *ASE'10*, pages 33–42, 2010.

[16] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 273–282. ACM, 2008.

[17] G. Canfora, L. Cerulo, and M. Di Penta. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 595–598. IEEE Computer Society, 2009.

[18] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97, Part 1:113 – 121, 2015. Special Issue on New Ideas and Emerging Results in Understanding Software.

[19] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 275–284. IEEE, 2014.

[20] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *ICPC'12*, pages 73–82, 2012.

[21] B. Fluri, M. Wursch, and H. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 70–79, Oct 2007.

[22] B. Fluri, M. Würsch, E. Giger, and H. C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, 2009.

[23] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich. Developers' code context models for change tasks. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 7–18, New York, NY, USA, 2014.

[24] M. Goeminne, A. Decan, and T. Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 353–357. IEEE, 2014.

[25] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *ISSRE'10*, pages 368–377, 2010.

[26] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 243–252. IEEE, 1994.

[27] M. Kamimura and G. Murphy. Towards generating human-oriented summaries of unit test cases. In *2013 IEEE 21st International Conference on Program Comprehension (ICPC)*, pages 215–218, May 2013.

[28] K. Kevic, T. Fritz, and D. Shepherd. Comogen: An approach to locate relevant task context by combining search and navigation. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 61–70, Sept 2014.

[29] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[30] M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, 2013.

[31] C. M. Kuok, A. Fu, and M. H. Wong. Mining fuzzy association rules in databases. *ACM Sigmod Record*, 27(1):41–46, 1998.

[32] B. Li, M. Grechanik, and D. Poshyvanyk. Sanitizing and minimizing databases for software application test outsourcing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 233–242. IEEE, 2014.

[33] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. Kraft. Automatically documenting unit test cases. In *ICST'16*, pages 341–352, 2016.

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

[35] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *37th IEEE/ACM International Conference on Software Engineering (ICSE'15) - Tool Demo Track*, pages 709–712. IEEE, 2015.

[36] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. How do developers document

database usages in source code? In *ASE'15 - New Ideas Track*, pages 36–41, 2015.

[37] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, Dec. 1987.

[38] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *Proceedings of the 30th international conference on Software engineering*, pages 451–460. ACM, 2008.

[39] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *ICPC'14*, page to appear, 2014.

[40] Microsoft. *Microsoft Application Architecture Guide.* Microsoft Press, 2nd edition, 2009.

[41] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 23–32. IEEE, 2013.

[42] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora. Automatic generation of release notes. In *FSE'14*, 2014.

[43] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker. Jsummarizer: An automatic generator of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 230–232. IEEE, 2013.

[44] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen, and T. N. Nguyen. idiff: Interaction-based program differencing tool. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 572–575. IEEE, 2011.

[45] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, pages 63–72, June 2012.

[46] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *38th International Conference on Software Engineering (ICSE 2016)*, pages 547–558, 2016.

[47] C. Parnin and C. Görg. Improving change descriptions with change contexts. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 51–60. ACM, 2008.

[48] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of renormalized relational databases. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 218–227. IEEE, 1996.

[49] D. Qiu, B. Li, and Z. Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 125–135. ACM, 2013.

[50] S. Rastkar. Summarizing software concerns. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 527–528, May 2010.

[51] S. Rastkar, G. Murphy, and A. Bradley. Generating natural language summaries for crosscutting source code concerns. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 103–112, Sept 2011.

[52] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Trans. Software Eng*, 40(4):366–380, 2014.

[53] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.

[54] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 43–52, 2010.

[55] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[56] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. Codes: Mining source code descriptions from developers discussions. In *22Nd International Conference on Program Comprehension*, pages 106–109, New York, NY, USA, 2014. ACM.

[57] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *ESEC/FSE'13*, 2013.

[58] A. T. T. Ying and M. P. Robillard. Selection and presentation practices for code example summarization. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–471, 2014.