Quality Assessment for Embedded SQL

Huib van den Brink
Utrecht University
The Netherlands
Email: hjbrink@cs.uu.nl

Rob van der Leek
Software Improvement Group
The Netherlands
Email: r.vanderleek@sig.nl

Joost Visser
Software Improvement Group
The Netherlands
Email: j.visser@sig.nl

***Abstract*—The access of information systems to underlying relational databases is commonly programmed using embedded SQL queries. Such embedded queries may take the form of string literals that are programmatically concatenated into queries to be submitted to the DBMS, or they may be written in a mixture of the syntax of SQL and a host programming language.**

The particular ways in which embedded queries are constructed and intertwined with the surrounding code can have significant impact on the understandability, testability, adaptability, and other quality aspects of the overall system.

We present an approach to tool-based analysis of the quality of systems that employ embedded SQL queries. The basis of the approach is the identification and reconstruction of embedded queries. These queries are then submitted to a variety of analyses. For example, we chart the relationships of queries to the surrounding code and, via the control and data flow of that code, to each other. Also, we define a suite of metric extractors for embedded queries.

Through a number of case studies, involving PL/SQL, Cobol, and Visual Basic, we show how the results of these analyses can be employed to make an assessment of various quality aspects related to the use of embedded queries.

I. Introduction

Many applications depend on a database system for persistent storage of data. The communication and relation with the database system is usually expressed via the Structured Query Language (SQL). The SQL queries are used to access and manipulate data in the database. To integrate the SQL queries into the program code, modern object-oriented systems tend to make use of a some kind of object-relational mapping (ORM) framework, such as Hibernate (for Java) or DataObjects.net (for C#). These frameworks offer the object-oriented programmer a degree of abstraction over the relational data. But many systems do not rely on the abstraction offered by an ORM framework and instead integrate queries into host programs by some form of *embedding*.

Some programming languages provide a dialect of SQL as sublanguage, allowing queries to be embedded in designated, keyword-marked block. For example, queries can be embedded in COBOL programs inside the `EXEC SQL` and `END-EXEC` keywords, as illustrated in Listing 1. Inside the query text, the variables of the host program can be referred to when prefixed with a colon.

```
MOVE       WD-BEG TO WD-BEG-SER-ID.
MOVE       WD-END TO WD-END-SER-ID.
EXEC SQL
           SELECT COUNT(*)
           INTO  :WD-COUNT
           FROM  TB110_SER
           WHERE T110_SER_ID
           BETWEEN :WD-BEG-SER-ID AND :WD-END-SER-ID
END-EXEC.
IF         SQLCODE = ZERO
  DISPLAY  "NUMBER OF SERIES PRESENT : " WD-COUNT
END-IF.
```

Listing 1. Embedding of an SQL query in COBOL, using extended syntax.

```
String qInterest = "SELECT EFFECT_YR_INTEREST "
        + " FROM " + db.TblSavings
        + " WHERE CREDIT_LIMIT_LOW <= " + LoanSum
        + " AND CREDIT_LIMIT_HIGH > " + LoanSum;
Connection conn = db.getConnection();
Statement st = conn.createStatement();
ResultSet rsQInterest = st.executeQuery(qInterest);
```

Listing 2. Embedding of an SQL query into Java, using string concatenation.

Other programming languages do not provide SQL as a sublanguage but instead queries are constructed as textual strings in the host language, before being passed to a library routine that executes them. A Java snippet that illustrates this approach is shown in Listing 2. String concatenation is used to glue query fragments together with host language variables.

With either form of embedding, data access can be coded according to many different styles and techniques. Different choices can be made, for instance, regarding the use of host variables, the distribution of functionality over queries and host programs, the variation points in pre-defined query templates, the reuse of these templates to construct different queries, and of course the structure of the queries themselves. These choices may have a significant impact on the quality of the system as a whole.

In this paper, we present an approach to tool-assisted quality assessment of the data access aspects of systems that employ embedded SQL. We are mainly interested in the *maintainability* characteristic of software product quality [1], but other characteristics, such as *reliability* and *efficiency* are also touched upon. Apart from the embedded queries themselves, we analyze the interconnections of the queries with the surrounding code of the host programs. The tool support has been developed as an extension to the Software Analysis Toolkit which the Software Improvement Group (SIG) employs for performing software risk assessments [2], [3] and software portfolio monitoring [4].

The paper is structured as follows. In Section II we explain the techniques we employ for distilling individual queries from program code. In Section III, we discuss how various relationships can be detected between host programs, embedded queries, and database tables. In Section IV, we defined measures to quantify quality aspects of embedded queries. Section V presents the results of case studies performed on several business critical software applications. Finally, Section VI discusses related work, and Section VII concludes.

II. Distilling queries from host programs

As indicated in the introduction, two main forms of embedding queries into host languages can be distinguished: queries constructed as strings of the host language; and queries located in designated blocks, using a syntactic extension to the host language. For each form we will explain how we distill queries from host programs to prepare for further processing.

A. Queries in extended syntax

To extract embedded queries from host programs written in the syntax of the host language extended with the SQL query syntax, two approaches can be taken: (i) merge the parsers of host language and query language, or (ii) cut query blocks out of host programs by lexical analysis, and apply the host language parser and query language parser separately.

The first approach is appropriate when high-quality grammars are available for both languages, in a compositional syntax definition formalism. This is the case, for instance, for SQLJ, which extends the Java syntax with SQL [5] and for PL/SQL, which extends SQL with a procedural programming language. For these languages, SIG has previously developed full context-free grammars in ANTLR [6] and SDF [7].

For the COBOL language, on the other hand, this a notoriously difficult precondition to satisfy. Many different dialects of COBOL exist, and it requires preprocessing before parsing. Efforts to recover a COBOL grammar suitable for reverse engineering purposes from language manuals have stopped short of dealing with preprocessor issues and do not include the SQL language extension [8]. Therefore, reverse engineering tools commonly rely on island grammars [9] or on non-context free parsing to obtain a degree of independence of dialect differences and language extensions.

For COBOL as well as Visual Basic 6 (VB6), we have opted for the last approach. This has allowed us to reuse our existing parsing and analysis functionality for COBOL, VB6, and SQL. The previously developed SQL grammar covers a variety of dialects, including DB2, PL/SQL, T-SQL, and the ANSI-92 standard.

Our analysis tools are programmed in Java, using the JJForester visitor generator [10] to generate syntax tree support from SDF grammars. We have used ANTLR for tokenization of host programs to prepare for embedded query extraction.

B. Queries constructed as strings

For the string concatenation case, when queries are constructed out of strings, many constructs have to be taken into account. The issue to be addressed when gathering queries in these situations, is the great freedom of expression. The ways the queries are specified are limited only by the creativity of the developer. However some constructs are more common than other ones. Therefore a trade-off is made in our framework to support the most common constructs, while allowing addition support for more specific structures with minimal effort.

Our query reconstruction mechanism builds up chains of queries objects. Each query object stores the source location from which it was distilled, the reconstructed SQL program text and corresponding AST, and information about the names and types of variables as far as can be reconstructed. Data and control flow is analyzed to arrive at the appropriate chaining of queries.

To start the reconstruction process, all strings present in the source code are collected and inspected. We then focus on those strings that start with one of the case insensitive keywords: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `ALTER`, or `DROP`. These strings are used as seeds for a backward flow analysis that leads us to the host program variable to which the query string is assigned.

Subsequently, the forward flow is followed from these variables is traced to gather all strings that represent the string fragments that are composed to form a query template.

Our reconstruction mechanism can be configured for different programming languages by supplying appropriate arguments for various string concatenation operators. For combining two strings for instance, the operators `+`, `||` or `&` are provided for respectively Java, PL/SQL and Visual Basic. The same holds for operators like `+=` and `&=`. Also the ending of a statement is passed as a parameter, i.e. ; for Java and _\n for Visual Basic.

To reconstruct the types of variables, we follow use-def chains to their declaration sites. Where available, database schema information is also taken into account. Some amount of constant propagation is used to discard variables that in fact always get instantiated to the same values. When conditionals occur in the control flow during string concatenation, several alternative queries are constructed.

While our approach leaves room for false positives, i.e. query parts not used as a query, in practise it turned out to be very effective. More details on how to reconstruct queries are given in [11].

III. Detecting relationships

Once embedded queries have been distilled from host programs, queries and programs can be analysed to detect several relationships between tables, host program, and queries. We discuss five kinds of relationships:

- Access: which programs access which tables?
- Duplication: how similar are queries to each other?
- Control: does the result of one query control execution of other queries?
- References: do programs establish references between tables (programmatic joins)?
- Deletion: do programs encode cascading deletes?

TABLE I
EXAMPLE OF A CRUD TABLE.

	Create	Read	Update	Delete
DONE		x		
LOG			x	
QUEUE		x		x
RATES		x		

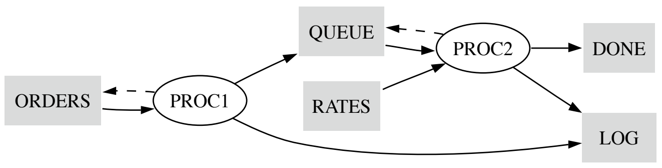

Fig. 1. Example of a CRUD graph. Dashed arrows indicate delete operations. Arrows from tables to programs represent selection, while arrows from programs to tables represent updates or inserts.

A. Access

A commonly used device to summarize the access of a program to a database is by the construction of so-called CRUD tables. CRUD describes the basic operations of a database, and stands for Create, Read, Update and Delete. Examples of these operations are, respectively: inserting new rows, selecting data out of the database, updating information, and deleting data. An simple CRUD table is shown in Table I. Thus, the four columns of a CRUD table stand for the four types of operation, while each row belongs to a different database table. CRUD tables can be constructed at different levels of abstraction: per query, per program unit (e.g. method or procedure), or per module (e.g. class or file or package). When data is accessed via views, the names of the underlying table should be resolved and entered into the CRUD table.

The CRUD information can also presented in a be different way, to give an abstract overview of the (potential) dataflow within a system. An example is presented in Figure 1.

B. Duplication

To detect queries that are variants of each other, the distilled queries are compared for equivalence. While some degree of duplication among queries is acceptable [12], abundance of (nearly) identical clones is detrimental to the system's maintainability.

In order to determine to what extent the same data resources or properties are used, a percentage of equivalence is computed for each pair of queries. Here equivalence means more than syntactic equality. The objective is to measure semantical equivalence as much as possible, for instance comparing the conditional parts of two queries in a commutative way. Comparisons are made in two steps. First, aliases are resolved to obtain a query AST that reflects the database structure. Second, the various query parts are compared. For instance when comparing `WHERE a = b AND c = d` with `WHERE a = b AND c = e` a 25% of inequality is registered for the WHERE part. The comparison is commutative, thus `a = b` is considered 100% equal to `b = a`. The duplication percentages of query parts are propagated to the query as a whole to obtain a single percentage per query.

```
SELECT MH_MN_SEQ.nextval
INTO df$rec.ID
FROM DUAL;

IF df$rec.ID IS NULL THEN
    BEGIN
        SELECT MH_MT_HUIZ.JND_GOEDGEKEURD
        INTO df$jnd.ID
        FROM DUAL;
    END;
END IF;
```

Listing 3. The result of one query controls the execution of another.

```
result = Database.Get("SELECT userId FROM User" _
    & " WHERE name = 'Alice'");

Database.Get("SELECT street FROM Address " _
    & " WHERE user_id = " & result.userId)
```

Listing 4. A programmatic join in Visual Basic 6.

C. Control

An `IF` statement condition relying on the result of a query, and which is regulating the execution of another query, imposes a relation of execution control of one query over another. An example is given in Listing 3. First a `SELECT` query is executed and the result is stored in `df$rec.ID`. The `IF` condition then determines if another query is to be executed or not.

Another example of one query controlling the execution of another is when a query result is used in a loop conditional, where the loop body contains another query. An example of such a control relation is given in [11].

D. References

References between tables are normally declared as foreign key relationships, which are then exploited to join multiple tables in a query. When queries are embedded into host programs, one can circumvent foreign key declarations and join queries by using the result of one embedded query as a parameter in a second embedded query. An simple instance of such as *programmatic join* is shown in Listing 4. When the result of the first query is a collection rather than a single row, then cursors or looping constructs of the host language are typically used to perform the join.

We detect programmatic joins and other referential relationships between tables by looking for certain patterns appearing in the query chains. The central focus for this analysis are the query results. When query results flow as arguments into the construction of other queries, a relationship may exist.

E. Programmatic cascading deletes

A common scenario with respect to foreign keys, is that when the item referenced is removed, the related information also should be removed. For example, when a user has a

```
-- Retrieve the user to remove
SELECT id
INTO usr_id
FROM User
WHERE name = 'Alice';

-- Remove the user
DELETE FROM User
WHERE name = 'Alice';

-- Remove the cars the user owns
DELETE FROM Car
WHERE usr = usr_id
```

Listing 5. A programmatic cascading delete.

car, and the user is removed out of the system, the related car also should be removed. This is solved by the construct of a CASCADE DELETE. When a CREATE TABLE or ALTER TABLE defines a column as being a foreign key by using the keyword REFERENCES, and thus referencing some primary key, the ON DELETE CASCADE option can be set. This option ensures that the rows referencing some primary key are removed when that primary key itself is removed.

For our analysis, this means that DELETE queries themselves do not contain information about if the delete command is cascaded or not. However, if the database scheme itself is present, these deletes related to the cascade behaviour are in any case being flagged as desirable situations.

Rather than relying on the CASCADE DELETE construct, declared in the database schema, some systems encode the same behaviour explicitly in the host programs. Our framework contains a mechanism for detecting such programmatic cascading deletes.

As an example of a typical programmatic cascading delete, consider the situation where a User can have multiple Cars, while a Car can only belong to one User. When the User is being removed, the Car must also be deleted. A programmatic way of achieving this is shown in Listing 5. As can be seen, only three points can vary here. The first DELETE query can reference either the id already retrieved, or state the same criterion as the SELECT query does. The second optional variation is the number of DELETE queries following the SELECT query, since other items referencing the User, in the same way the Car does, can be removed in the same way. The last variation point is the order of the DELETE queries. They can go in any order, as long as the SELECT precedes them.

A similar pattern can be detected in the situation where a cascading delete is programmed for a many-to-many relationship. Assume, for example, that a User can have multiple Addresses, while an Address can belong to multiple Users. In a database scheme this many-to-many relation is typically represented with a cross-reference table, as illustrated in Fig. 2. The scheme contains a user, which exists of a name, and the address that is a street name and the number of the house. The usr and adr in the cross reference table UserAddress respectively reference the id of the user and the id of the address. When an Address is being deleted, also all related

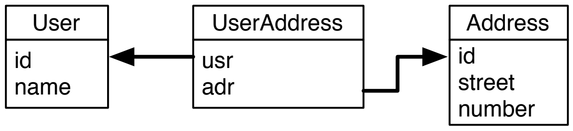

Fig. 2. Database schema for many-to-many relationship.

```
-- Find address to delete
SELECT id
INTO adr_id
FROM Address
WHERE street = 'Great Ocen Road'
AND nr = 13;

-- Retrieve all users living at that address
FOR usr_id IN
(
   SELECT usr
   FROM UserAddress
   WHERE adr = adr_id
)
LOOP
   -- Get the nr of addresses the user has
   SELECT COUNT(*)
   INTO adr_amount
   FROM UserAddress
   WHERE usr = usr_id;

   -- Remove users only if the user
   -- no longer has any address
   IF adr_amount < 2 THEN
      DELETE FROM User
      WHERE id = usr_id
   END IF;
END LOOP;

-- Remove foreign keys
DELETE FROM UserAddress
WHERE adr = adr_id;

-- Remove address
DELETE FROM Address
WHERE id = adr_id;
```

Listing 6. Delete a many-to-many relation.

Users should be deleted out of the system, but only if they no longer have any Address left.

Listing 6 shows how this cascading delete can be achieved programmatically. First the id of the address to be removed is obtained. Then for each user living at that address the number of associated addresses is counted. If the user only has one address, which is going to be removed, the user itself is removed. Then, all references to the address are removed. Finally the address itself is removed. 'Thus, first three SELECT queries are encountered placing their results in the variables adr_id, usr_id and adr_amount. Then the three DELETE queries use the first two parameters. The third parameter only is used for control flow. All queries reference only one table.

Generalizing these observations, the following pattern describes a programmatic cascade delete operation: A delete query using data from the result of a select query, followed or preceded by one or more other delete queries (that not

TABLE II
NUMBER OF OPERATIONS PER FILE.

	SELECT	INSERT	UPDATE	DELETE	CREATE	ALTER	DROP
TCDMUY07.COB	14	11	25	6	0	0	0
PJT_PARAM.trg	0	0	0	0	2	0	0

necessarily use data from a select query), implies an operation performed by the host language fulfilling the semantics of a cascade delete. This pattern is detected in the chains of embedded queries that have been distilled..

IV. QUANTITATIVE QUERY MEASURES

To support an objective estimation of the quality of systems with embedded SQL queries, we have defined and implemented a series of quantitative measures. We will discuss a representative selection of these. Mapping such quantitative measures to quality aspects as defined for example by the ISO 9126 quality model [1] is discussed elsewhere [3].

A. Occurrence measures

1) Number of queries: We count the number of distilled embedded queries per source code file. Aggregated numbers per package, system layer, or other group of files can also be presented. These numbers provides a global insight into the embedded query usage of the system.

Files situated in the persistence layer are expected to contain queries, while for the domain and graphical user interface layer this is highly undesirable. Another indication is being retrieved about how centralized the handling of the queries is. Having the queries about anywhere in the system indicates a poor separation of concerns at architectural level, which does not benefit maintainability.

2) Number of operations: Embedded queries can be categorized by the type of operation they perform. This categorization can be defined in several ways. We present the number of each query operation type per file, as shown in Table II. The number of operations can also be aggregated per group of files such as those belonging to a package or system layer.

The create, alter, and drop operations should be expected to occur in lower numbers and more centralized than the select, insert, update, and delete operations.

3) Number of string queries: For languages that allow embedded queries by means of an extended syntax, such as PL/SQL and COBOL, we measure how many queries are constructed by concatenation of strings. We expect this number always to be 0, but as we will see in Section V, this is not always the case.

B. Variation points

Embedded queries are typically templates with variation points indicated by host variables. These variation points give rise to several measures.

```
SELECT name, street, phone
INTO :name, :street, :phone
FROM User
WHERE id = :uid
```

Listing 7. A query with 3 result variables and 1 input variable.

```
CURSOR cur
IS
   SELECT id, name
   FROM   User;
```

Listing 8. A cursor query with 2 select items but only one result variable.

1) Number of result variables: This measurement counts the number of result variables that are used to store the query result in. For example, the query in Listing 7 contains three result variables, i.e. the `:name`, `:street` and `:phone`. This measure is not always equal to the number of select items, i.e. the columns selected for the query result, in particular when cursors are used. The PL/SQL example of Listing 8 has two select items, but only one result variable, *viz* the cursor `cur`.

Excessively high numbers of result variables may indicate queries that are difficult to maintain.

2) Number of (unique) input variables: Input variables may be used more than once in a query, so we can both count the unique input variables, or just the number of holes in the query, without correcting for double occurrences.

We find that the non-corrected measure is a better indicator for query complexity, since double occurrences of input variables make queries less understandable and adaptable.

3) Number of variation points per query part: Variables in a query and its subqueries can be categorized by the query part in which they occur. We distinguish the following query parts, ordered from common practise to potentially harmful:

- Variable criterion - `WHERE` clause
- Variable additional criterion - `HAVING` clause
- Variable result column - `SELECT` item
- Variable grouping - `GROUP BY` cause
- Variable table - `FROM` clause

Table III shows the measurement results for some of Listings in this paper.

Using variables in the `WHERE` clause is common practise. Variability in the `HAVING` clause is uncommon, but acceptable. The selected columns and the columns by which grouping takes place are usually fixed. Finally, varying the

TABLE III
NUMBER OF VARIATION POINTS PER QUERY PART

	WHERE	HAVING	SELECT	GROUP BY	FROM
Listing 1	2				
Listing 2	2				1
Listing 4	1				
Listing 6	5				

table queried is almost always undesirable, because such variation points are more difficult to understand and harder to test.

Inspection of the top scoring queries in each column can rapidly reveal problematic queries in an application.

C. Measures related to query structure

1) Number of tables used: A simple measurement for complexity is performed by counting the number of tables used by each query. The more tables used, the more complex the query gets, because of the many joins and nestings they are forced to use. Instead of inspecting the `FROM` clause, this information is derived from the previously constructed CRUD tables (see Section III-A).

Note that complex queries, in terms of number of tables used, are not necessarily undesirable. In fact, if the alternative to query complexity is to shift complexity to the host program, e.g. by introducing programmatic joins, complex queries may be preferable from the point of view of performance and maintainability.

2) Number of nested queries: A nested query is an SQL `SELECT` statement that is placed in the predicate of any other SQL statement. We count the number of nested queries per embedded query location.

For the usage of nested queries, each level deeper implies a significantly increased complexity. The result of the nested queries are transient, which makes the entire query harder to understand and to test. This increased complexity however could be taken for granted when the performance is benefiting from this construct. The query optimizer of the DBMS may be able to apply particular optimizations to the query with nested subquery, but not to the two queries separately. Also, removing nested queries may just result in shifting complexity to the host language.

3) Number of `UNION` queries: This measurement counts the number of used `UNION` and `UNION ALL` keywords.

Union queries are potentially detrimental to performance, and should be used sparingly.

4) The number of joins used: To obtain an exact count of the numer of joins in embedded queries, schema information is needed [11], but this information is not always available. A rough count can be obtained simply by counting the number of `JOIN` keywords present. Alternatively, the number of tables referenced in the `FROM` clause can be taken as a substitute measure (see IV-C1). After all, the tables specified there normally are connected in some way in the criterion part of the `SELECT` query.

D. Measures of relations between queries

1) Equality: As explained in section III-B, for each query the equality with respect to all the other queries is computed. For each pair of two queries, the equality is expressed as a number between 0% and 100%. Based on these equality numbers for query pairs, we derive two measures to quantify individual queries.

- Maximum: The first measure takes for each query its highest equality percentage with respect to some other query. With this measure, the queries that show the most resemblance to others obtain the highest scores.
- Sum: The second measurement sums up all equality percentages for a given query. With this measures, queries that present similarity to many other queries score highest.

Suppose a system contains (i) a query that is 100% identical to some other query, and 0% identical to all others, and (ii) a query that has 10% similarity to 15 other queries. When taking the maximum, the first query scores higher (100 vs. 10), but when taking the sum, the first query scores lower (100 vs. 150).

High scores for equality measures may indicate a lack of internal reuse in the application. Rather than repeating the same or similar queries in many different places, a single, parameterized query should be embedded in a subroutine that is subsequently invoked from different host programs.

2) The number of depending queries: Multiple `INSERT` or `DELETE` queries can use data obtained by one `SELECT` query as is explained in the Sections III-C (control dependencies), III-D (programmatic joins) and III-E (programmatic cascading deletes). The analysis results these sections constructed, is used by this measurement of dependent queries. For the `SELECT` queries it is counted how many other queries there are, that use its query result, creating a dependency on the particular `SELECT` query.

The queries not producing a result, i.e. not retrieving information out of a database, do not have this type of dependency, because they just perform actions. Their result, i.e. side effecting, is not used directly as input for other queries.

V. Case study

The presented tool support is suitable for analysis of queries embedded in many different programming languages, including PL/SQL, COBOL, Visual Basic 6, and Java. Furthermore, the tool support is easily extensible to other language.

We performed case studies on the following business critical software systems:

- PL/SQL source of a bank
- PL/SQL source of an energy supplier
- COBOL source of a bank
- Visual Basic 6 source of a telecom company

The two banks mentioned here refer to distinct banks. In the next two sections the results of the PL/SQL source of the energy supplier and those of the COBOL source will be discussed briefly. A more complete overview can be found elsewhere [11].

A. PL/SQL system of energy supplier

For the energy supplier, Table V lists the results of the occurrence measurements for the entire system, and for three files separately. Quite some queries were present and, as expected, most of them were of the type `SELECT`. Having

TABLE IV
MEASUREMENTS OF THE PL/SQL SOURCE

file	line	res vars	vars	nesting	union	join	tables	equality	dependencies
dump444.out	178	**59**	2	0	0	0	1	97%	0
dump359.out	166	0	**66**	1	0	0	11	9%	0
dump249.out	136	0	20	**4**	0	0	1	69%	0
dump98.out	2692	0	10	0	**2**	0	2	81%	0
dump65.out	832	0	33	0	0	0	**18**	653%	0
dump296.out	3349	0	17	0	0	0	2	**3302%**	0
dump218.out	1353	2	3	0	0	0	1	186%	**37**
dump65.out	1369	0	2	0	0	0	2	351%	0

TABLE V
OCCURRENCE MEASURES FOR A PL/SQL SYSTEM.

	amount	SEL	INS	UPD	DEL	string
All	5475	3427	787	912	245	104
dump221.out	59	22	4	7	2	**24**
dump222.out	**211**	71	7	**71**	**56**	6
dump208.out	206	**101**	**40**	28	22	15

TABLE VI
OCCURRENCE MEASUREMENTS FOR A COBOL SYSTEM.

	amount	SEL	INS	UPD	DEL
All	496	282	65	115	34
x.COB	**59**	**24**	9	21	5
y.COB	56	14	**11**	**25**	6
z.COB	50	19	**11**	11	**9**

TABLE VII
MEASUREMENTS OF THE COBOL SYSTEM.

file	line	res vars	vars	nests	tbls	eq
x.COB	3485	**48**	1	0	1	9%
x.COB	3601	0	**49**	0	1	4%
x.COB	1158	23	5	0	**5**	15%
w.COB	1502	13	2	0	2	**34%**

more UPDATE than INSERT queries is also unsurprising. No CREATE, ALTER or DROP queries were found, indicating that no database initialization code was included in the delivered source code. A surprising finding is that 104 queries in this PL/SQL project are constructed out of strings, rather than in the syntax that the language offeres for this purpose.

Various other measurements are displayed in table IV, where each row represents a single query. Queries using large numbers of variables, as present in this project, normally imply INSERT queries that obtain information from the surrounding source. The source at that point then is likely to point directly to the complex parts of the software, where lots of state is involved. This indication also applies to the queries dealing with many result variables, leading to parts consuming much information.

While this project contains a large number queries, the biggest duplication in a single query (at line 3349 in file dump296.out) was 3302%, meaning that the query is equal to 33 other queries or half equal to 66 other queries. Thus, a high degree of duplication is present among the queries for this application, indicating that improving this aspect would likely increase its quality.

The nesting metric indicates that complex queries are present. This indication is confirmed by the queries that use a large number of tables. For the query at line 832 in 'dump65.out' this is remarkable because this query is not being composed out of queries combined with the UNION construct. Refactoring this query therefore probably will simplify the application. No join keywords were used, and from the point of view of the surrounding source, the number of queries depending directly on a single query is at maximum 37. This means that changing one query may influence the behaviour of 37 other queries, thus limiting adaptability and testability.

The last entry of the table represents a SELECT query at line 1369 in 'dump65.out'. This query contains two variables: one variable in the selected items and one in the WHERE clause. This indicates a malformed database structure or a bad practise in the host language, as explained in section IV-B3. In total this was encountered twice in this project.

B. COBOL system of a bank

The occurrence metrics for the financial COBOL system are shown in Table VI. This first overview shows that the different queries are well centralised, because 33.3% of all queries are contained in just three files. Although this led to large files, the queries at least are not scattered throughout many files.

The other observation is that 56.8% of all queries are SELECT queries, 23.3% are UPDATE operations, 13.1% are INSERT and 6.9% are DELETE queries. Per file the composition of these query types hardly differ, thus showing a normal usage of the different query operation types. The measurements that turned out relevant for this project are shown in Table VII In the COBOL project quite some queries were used, but the queries themselves proved to be simple and straightforward. No nesting of queries occurred, no unions were present, only a handful of queries used more than two tables, while the most resembling queries are only 34% equal. In order to achieve this simplicity, the programmers used many, uniquely named variables to handle the data throughout the application. So the work was shifted from the queries and the DBMS to the host language. This leads to moving data around quite a bit in the host language, meaning unnecessary indirection.

VI. RELATED WORK

Goldberg and Brass have studied the issue of software quality specifically for SQL queries [13]–[16]. Their work does not cover the interaction of embedded queries with host programs, but deals with isolated queries. Though their work also applies to embedded queries, we have focussed on the additional complications that arise from their embedding.

Gould *et al*. have studied semantic errors in dynamically created queries, based on a Java byte-code analysis [17]. This work differs from our approach in the sense that it is more restrictive, because they initially detect the locations that access the database. On the other hand, our approach is able to work with unknown database access frameworks and mechanisms. Furthermore, we are able to handle all common SQL dialects, and our analysis can be configured to accept virtually any imperative programming language ahs host language. Our analysis is performed at source level, but does not require a full set off compilable sources to be available.

Henrard has studied the extraction of embedded queries from host programs in the context of database reverse engineering [18]. He describes techniques for performing data structure extraction. The objective of our approach is quality assessment rather than reverse engineering, and we perform a wider set of analyses.

VII. Conclusion

A. Contributions

We have presented an approach to tool-based analysis of the quality of systems that employ embedded SQL queries. The approach covers two forms of embeddings: where queries are constructed dynamically as string values, and where queries are expressed via an extended host language syntax.

We have presented tool support that is able to handle embedded queries in a wide range of host languages, such as COBOL, PL/SQL, Visual Basic 6, and Java. The support for distillation of queries from source programs can with little effort be configured for more host languages.

We have described the analyses we implemented for detecting relationships among queries, host programs, and tables, such as computing query similarities, extracting programmatic joins and cascading deletes, and summarizing data access in CRUD graphs.

A suite of metrics have been defined and implemented for quantifying quality aspects of systems that contain embedded SQL queries. The metrics quantify occurrence, variation points, structure, and interrelations of queries.

Several case studies have been conducted on business-critical software systems, and the result and their implications for software quality have been presented. For all the analysed software projects, the experiment revealed, after manual inspection, that the obtained values actually do reflect the situation, as it is present in the software application.

B. Future work

The tool support can be configured and extended for further host languages. Also, more metrics can be defined and implemented.

The tool support was designed to give usable results, also when no database schema is present. Additional work can be done to improve accuracy of several analyses when indeed the schema is not available. A possible line of attack would be to reconstruct as much as possible of the schema, based on the queries.

Our work is specific for SQL and relational databases, but for other types of databases, such as IMS, a similar approach could be used. IMS queries are also commonly created by a form string concatenation. We think the current implementation can be extended to cover embedded IMS with little effort.

References

[1] ISO, *ISO/IEC 9126-1: Software Engineering - Product Quality - Part 1: Quality Model*. Geneva, Switzerland: International Organization for Standardization, 2001.

[2] A. van Deursen and T. Kuipers, "Source-based software risk assessment," in *ICSM '03: Proceedings of the Int. Conf. on Software Maintenance*. IEEE Computer Society, 2003, p. 385.

[3] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring mainainability," in *QUATIC 2007*, 2007, to appear.

[4] T. Kuipers and J. Visser, "A tool-based methodology for software portfolio monitoring." in *Software Audits and Metrics, Proceedings of the 1st Int. Workshop on Software Audit and Metrics, SAM 2004, In conjunction with ICEIS 2004, Porto, Portugal, April 2004*, M. Piattini and M. Serrano, Eds. INSTICC Press, 2004, pp. 118–128.

[5] ISO, *ISO/IEC 9075-10: Information technology - Database languages - SQL - Part 10: Object Language Bindings (SQL/OLB)*. Geneva, Switzerland: International Organization for Standardization, 2003.

[6] T. Parr, *The Definitive ANTLR Reference, Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[7] E. Visser, "Syntax definition for language prototyping," Ph.D. dissertation, University of Amsterdam, 1997.

[8] R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Software—Practice & Experience*, vol. 31, no. 15, pp. 1395–1438, December 2001.

[9] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2001, pp. 13–22.

[10] T. Kuipers and J. Visser, "Object-oriented tree traversal with JJForester," in *Electronic Notes in Theoretical Computer Science*, M. van den Brand and D. Parigot, Eds., vol. 44. Elsevier Science Publishers, 2001, proceedings of the Workshop on Language Descriptions, Tools and Applic ations (LDTA).

[11] H. J. van den Brink, "A framework to distil SQL queries out of host languages in order to apply quality metrics," Jan. 2007, MSc. thesis.

[12] C. Kapser and M. W. Godfrey, "'cloning considered harmful' considered harmful." in *13th Working Conference on Reverse Engineering (WCRE 2006)*. IEEE Computer Society, 2006, pp. 19–28.

[13] C. Goldberg and S. Brass, "Semantic errors in SQL queries: A quite complete list," in *Fourth Int. Conf. on Quality Software (QSIC'04)*. Martin-Luther-Universität Halle-Wittenberg, 2004, pp. 250–257.

[14] ——, "Proving the safety of SQL queries," in *Fifth Int. Conf. on Quality Software (QSIC'05)*. Martin-Luther-Universität Halle-Wittenberg, 2005.

[15] ——, "Detecting logical errors in SQL queries," in *16th Workshop On Foundations Of Databases (2004)*. Martin-Luther-Universität Halle-Wittenberg, 2004.

[16] C. Goldberg, S. Brass, and A. Hinneburg, "Detecting semantic errors in SQL queries," Martin-Luther-Universität Halle-Wittenberg, Tech. Rep., 2003.

[17] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *26th Int. Conf. on Software Engineering (ICSE 2004)*, University of California, Davis. ACM Press, Sep. 2004.

[18] J. Henrard, "Program understanding in database reverse engineering," Ph.D. dissertation, Facultes Universitaires Notre-Dame de la Paix namur, Aug. 2003.