Towards a Safe Method for Computing Dependencies in Database-Intensive Systems

Csaba Nagy, János Pántos, Tamás Gergely, Árpád Beszédes
University of Szeged
Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
{ncsaba,pantos,gertom,beszedes}@inf.u-szeged.hu

***Abstract*—Determining dependencies between different components of an application is useful in lots of applications (*e. g.,* architecture reconstruction, reverse engineering, regression test case selection, change impact analysis). However, implementing automated methods to recover dependencies has many challenges, particularly in systems using databases, where dependencies may arise via database access. Furthermore, it is especially hard to find safe techniques (which do not omit any important dependency) that are applicable to large and complex systems at the same time. We propose two techniques that can cope with these problems in most situations. These methods compute dependencies between procedures or database tables, and they are based on the simultaneous static analysis of the source code, the database schema and the SQL instructions. In this paper, we quantitatively and qualitatively evaluate the methods on real-life data, and also evaluate them on some of their potential applications.**

***Keywords*-Program dependencies, program analysis, databases, SEA relations, CRUD matrix.**

I. Introduction

Analyzing program dependencies is helpful in many different phases of the software development process, such as in architecture reconstruction, reverse engineering, regression test case selection, and change impact analysis. A certain class of automated methods for computing program dependencies uses static analysis, usually by employing call, control flow, and data flow information. The granularity of these methods ranges from the basic block level through procedure[1] and class levels to module and system levels, each granularity level having potential application areas.

Static analysis has its own pitfalls, though: in many situations, we have to decide if want a safe result (meaning that no important dependency is missed) or a concise one (focused on the certain dependencies only). On the one hand, a high-level analysis can be made safe by including all the possible dependencies that might arise during the executions of the code being examined, but this might result in many fake dependencies, thus the result is not very useful. On the other hand, a detailed low-level static analysis might find just the relevant dependencies, but have high computation costs and make it impractical on real-size systems. It is also possible that safety is sacrificed to make a method faster, and heuristics are used instead of detailed computations that might miss some rare, but important, dependencies.

In addition, bigger information systems are usually heterogeneous in the sense that more than one programming language and technology are used in them. The most common situation is that relational databases are used behind procedural programs. In this case dependencies may (and will) arise through the database, which are usually not detected by static analysis tools. It is possible to extract some SQL instructions that access the databases, but a static analysis is usually not enough to recover all of them exactly (consider, for example, SQL query-strings assembled at execution time).

Here we propose two methods that compute dependencies at the procedure level, are applicable on real-size systems, and when properly applied, can provide safe results. The first method is based on Static Execute After/Before relations (SEA/SEB [1]), which uses the static call- and control flow graph and a lightweight interprocedural analysis of the system. The second method analyses the embedded SQL statements of the code and the database schemas to discover dependencies via database access. It computes $CRUD$-based dependencies [2], [3] between the SQL statements, and propagates them to the procedure level.

We applied these methods on program code obtained from one of our industrial partners. Here architecture reconstruction and test coverage evaluation were performed. The main contributions of the paper are:

- the application of a $CRUD$-based Usage Matrix for dependency analysis between program elements, which we think is a novelty (Usage Matrix is a common way of recovering dependencies between client applications and database tables);
- adapting SEA relations to recover dependencies in database-intensive systems;
- applying a combination of these two methods and empirically evaluating them on real-life data.

The paper is organized as follows. In Section II we give an overview of related work. We provide a detailed description of the algorithms in Section III, then in Section IV we provide a quantitative and qualitative analysis of the methods, and we present some possible application scenarios. Lastly,

[1]In this paper the term *procedure* means any general procedure. A stored procedure of a database system will be called a *stored procedure*.

in Section V we discuss our results, draw some pertinent conclusions and suggest some ideas for future study.

II. Related Work

The System Dependence Graph (SDG), which describes traditional software dependencies (e. g. control and data dependencies) between different source elements of a system, is a common tool for software dependency analysis [4]. The construction of the SDG has been a real challenge for the slicing community [5], [6] for a long time, because – depending on the source code being analyzed – one has to tackle a range of problems like context sensitivity, pointers, and threads. This is why many studies have focused on software dependencies. However, only a few of them concentrate on dependencies arising via database access, e. g. Sneed *et al.* in [7].

Another goal of identifying these kind of dependencies is the impact analysis of schema changes. Maule *et al.* published a technique [8] in this area which is based on a k-CFA algorithm for extracting SQL queries. They use the SDG to identify the impact of relational database schema changes upon object oriented applications. Gardikiotis *et al.* [9] use a DA (Database Application) specific version of the PDG (Program Dependence Graph) to perform a static program slicing analysis. They extended the PDG with special pseudo nodes for SQL statements and their relations to the database schema.

Another potential application of the identified dependencies is test case selection, and testing database applications in general. Haraty *et al.* introduced a method for regression test selection for database applications [10], [11]. Besides using traditional control and data dependencies, they introduced the notion of a dataflow analysis method that uses database interactions and it is based on identifying the usage of table columns. This idea is similar to our $CRUD$ relation idea. Their method works on SQL's PSM extension (Persistent Stored Modules, e.g. Pl/SQL). Thus they do not need to deal with the problems of dynamically concatenated SQL queries, which is important if one would like to generalize this technique to other procedural or object-oriented languages.

A $CRUD$ or Usage Matrix is also useful for system understanding and quality assessment purposes. Deursen *et al.* [2] used it in their work to identify conspicuous table usage in COBOL applications. For instance, tables employed just for retrieving data and top used tables. Brink *et al.* [3] used Usage Matrix to calculate metrics for applications with embedded SQL.

The dependencies computed from the source code can be applied to support data reverse engineering too. Henrard *et al.* published papers in this area [12], [13] and evaluated different dependency analysis techniques (variable dependency graph, program slicing) via database access positions applied to dependency elicitation.

Most of the above-mentioned methods heavily depend on the extraction of SQL statements from the source code. Finding solutions for this problem has also been a big challenge for researchers and many approaches have been proposed for static and dynamic analysis techniques as well. Most of the static methods apply string analysis techniques [14], [15], as we do too, but ours does not implement control or data flow analysis in order to keep the extraction of query strings fast and scalable for large systems. Hainaout *et al.* recently published two papers [16], [17] which describe different techniques and also some applications in this area. Cordy *et al.* published a number of papers [18], [19] describing the TXL language and its applications including embedded SQL parsing. Their TXL language and agile parsing technology could also be used to extract embedded SQL statements and/or to parse incomplete SQL statements. However, they agree that applying their agile technology would require more resources than 'normal' parsers.

III. Methods

A. Overview

1) Motivation: The main idea behind our methods is that in data-intensive systems many dependencies between the program elements arise via database accesses, which are usually not recovered by traditional dependency algorithms.

For instance, suppose that one procedure (called f) inserts data into a table T and later during execution another procedure (called g) reads data from table T to execute a complex algorithm which takes the same data as its input parameter. Obviously, when we modify the first procedure and perform a change impact analysis, we must examine the second procedure too. This is the only way to make sure that our algorithm implemented in g still works as expected. Sometimes traditional algorithms are able to determine whether g depends on f ($f \rightarrow g$), but there are many situations when $f \rightarrow g$ will not be detected. For example, a simple call or a control flow analysis will be able to recover the $f \rightarrow g$ relation if there is a chain of call or control flow dependencies from f to g. However, suppose that our application is multi-threaded and f and g run in different threads, furthermore f never calls directly or transitively g. In this case, the relation between the two procedures will appear neither in the call graph nor the control flow graph of the system.

In another common situation, let us suppose that the above-mentioned f and g are called in a procedure body one after the other; but g is never called directly or indirectly from f. The traditional call graph-based methods will not recover their dependency relation either. Therefore, it would not be safe to perform a change impact analysis in this case.

2) Methods: Here we propose two new methods for recovering dependencies in database-intensive systems that complement traditional software dependencies. One is a previously published algorithm that determines Static Execute

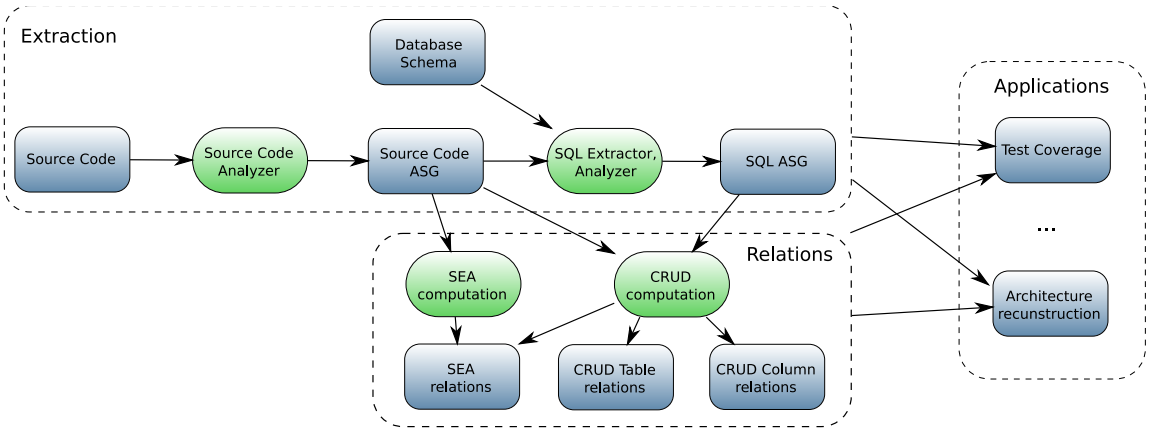

Figure 1. Methods overview. The main parts of the methodology are the steps for extracting source information (analyzing the source code, extracting and analyzing SQL instructions) and the steps for computing SEA and $CRUD$ relations. The collected information can be used for a variety of purposes.

After or Static Execute Before (SEA/SEB) [1] relations and the other is based on the Usage Matrix ($CRUD$ for short) [2], [3].

Dependencies computed by both SEA/SEB and $CRUD$ describe special relations which are not recognizable by traditional software dependencies, and when applied suitably, can provide safe results at a reasonable cost, even in the case of large database-intensive systems.

An overview of our analysis system is given in Figure 1. Both algorithms take their input from the Abstract Semantic Graph (ASG) extracted from the source code, but in the case of the Usage Matrix we also need to extract the SQL instructions embedded in the source code and analyze them separately. The outputs of the algorithms are a list of computed relation pairs: procedure-to-procedure, procedure-to-table, table-to-table, and column-to-column pairs. These relations can be used in application areas like architecture reconstruction and test coverage measurement.

B. $CRUD$ relations

1) Introduction: After a successful extraction and analysis of embedded SQL statements, it is possible to determine the relations between the program statements and between the accessed tables or columns. To achieve this, an analysis of the database schema is required along with an analysis of extracted SQL statements. Should the schema source code not be available, it can be obtained from the database management system. After retrieving this additional piece of information, the computed relations can be used to construct the Usage Matrix of the system. This matrix describes the usage relations between the procedures and between the tables of the system. In our case, the relations are the basic $CRUD$ (*Create*, *Retrieve*, *Update*, *Delete*) operations. Namely,

- `INSERT` statements *create* data in their target table;
- a typical way of *retrieving* data from a table is a `SELECT` statement, but any other statements (even an `INSERT` or `DELETE` statement) can retrieve data from tables;
- `UPDATE` statements *update* data in their target table;
- `DELETE` statements *delete* data from their target table.

A typical $CRUD$ matrix can be seen in Figure 2.

	Customers	Rentals	Cars
NewCustomer	C		
CarRental	R	C	R
AddressModification	RU		
CarCrash			D

Figure 2. A typical $CRUD$ matrix. NewCustomer inserts data into the Customers table. CarRental reads data from the Customers and Cars tables, and inserts data into the Rentals table. AddressModification retrieves and updates the Rentals, and CarCrash deletes data from the Cars table.

The same information can also be presented in a graph called a *Usage Graph*, which shows the different kinds of relations between the procedures and tables. A typical Usage Graph can be seen in Figure 3. Solid and dashed arrows with different directions represent different relation types.

2) Relations between procedures via table access: The computed $CRUD$ matrix can be used to determine relations on different levels of the system. We say that procedures are related by $CRUD^T_{PP}$ if they share at least one accessed table.

Formally, for f and g procedures $(f, g) \in CRUD^T_{PP}$ if and only if $\exists T$ table which is accessed by both f and g.

This sort of relation has no direction, hence it is symmetrical: $(f, g) \in CRUD^T_{PP} \Leftrightarrow (g, f) \in CRUD^T_{PP}$. It is not necessarily transitive because it may happen that $(f, g) \in CRUD^T_{PP}$ because they access only T_i and $(g, h) \in CRUD^T_{PP}$ because they access only T_j, but $(f, h) \notin CRUD^T_{PP}$ because they do not share an accessed table, that is, $i \neq j$.

A visual representation of these relations can be seen in Figure 4.

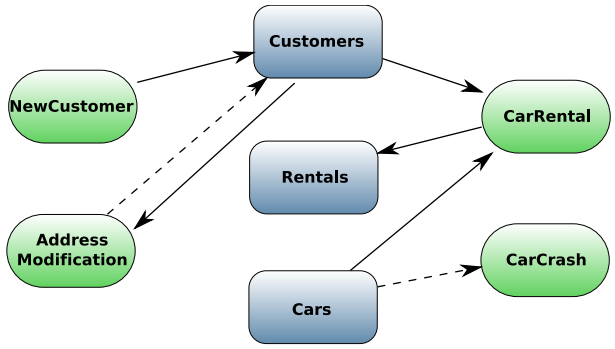

Figure 3. A typical $CRUD$ graph. The tables are in the centre of the figure and the procedures are in the oval shapes on both sides of the figure. The arrows from the tables to the procedures represent data retrieve operations, while the arrows from the procedures to the tables represent updates. The dashed arrows from tables to procedures represent delete operations and the dashed arrows from procedures to the tables represent create operations.

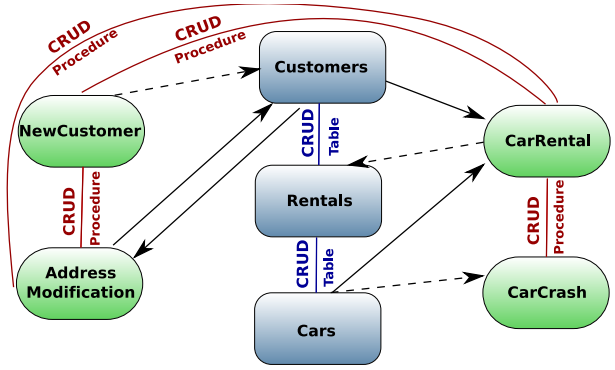

Figure 4. Typical $CRUD$ relations between procedures and between tables.

This relation can be computed in a safe way even though the SQL extraction may not recover the exact SQL statements for each embedded SQL string in the source code. When an extracted SQL instruction contains an unrecognized fragment in the place of a table identifier, a conservative approach inserts the procedure into the Usage Matrix as it would be related to all the tables of the database. For example, in the case of a procedure containing an SQL command like the one in Figure 5, a conservative approach relates the procedure to each procedure that access any table in the system.

Here, notice that the most common reason for unparsable SQL strings is that they are sometimes constructed with code fragments in a position where it makes the full statement syntactically incorrect. However, there are several other reasons, which we will elaborate on in Section IV-D.

3) Relations between procedures via column access: $CRUD$ operations can be lowered to the database column level by considering exact column access instead of a table access. The idea behind this low-level consideration is that even if a procedure modifies a table and another one reads

```
SELECT firstname, lastname
  FROM @@customer_table@@
  WHERE firstname
    LIKE('%@@name@@%');
```

Figure 5. Sample code of an extracted SQL command where the table name is determined by a variable.

data from the same table, there is no data dependency between them unless they modify and read the same record(s) of the table. However, determining the accessed record(s) of the table is not possible via a static analysis. Nevertheless, it is still possible to find the accessed columns of the table, and the scope of the dependency relation can be narrowed down. Harraty *et al.* suggest this level of granularity in [10].

We shall define $CRUD$ operations for the relations between procedures via column access like so:

- `INSERT` statements *create* data in all the columns of their target table;
- `SELECT` and those statements which do not modify data, *retrieve* data from columns which are accessed only for reading. The asterisk in a `SELECT` means a data retrieval for all the columns of the corresponding tables;
- `UPDATE` statements *update* specified columns of their target table;
- `DELETE` statements *delete* data from all the columns of their targeted table.

Formally, for f and g procedures $(f, g) \in CRUD^{C}_{PP}$ if and only if $\exists C$ column which is accessed by f and g.

$CRUD^{C}_{PP}$ can be also computed via a conservative approach. However, in procedural languages where SQL commands are constructed in a dynamic way, the noise of such a conservative method would result in too many false positive relations.

4) Relations between tables or between columns: The Usage Matrix can be used to determine relations between tables ($CRUD_{TT}$) or between columns ($CRUD_{CC}$) of the system too. This approach is based on database and program reverse engineering [13]. The idea behind it is that there are many kinds of dependencies between table columns that are not recognizable by the traditional database reverse engineering techniques that only analyze the database of a system. Certain kinds of dependencies require taking into account the embedded queries in the source code as well. Columns or tables accessed by the same procedures are related to each other and these relations must be considered when carrying out data reverse engineering (i.e. modularization).

Similar to $CRUD^{T}_{PP}$, the $CRUD_{TT}$ and $CRUD_{CC}$ relations can be defined as follows: for t and q tables (columns) $(t, q) \in CRUD_{TT}$ ($CRUD_{CC}$) if and only if $\exists P$ procedure which accesses both t and q.

These kind of dependencies can be also recovered by

using the Usage Matrix. However, a conservative implementation should be applied with care as an unrecognized code fragment will mean that all the tables or all of the columns in the system will be related to each other.

C. SEA/SEB relations

1) SEA/SEB in general: The Static Execute After/Before dependencies and an algorithm for their computation were previously published by Jász *et al.* [1]. According to their definition $(f, g) \in SEA$ if and only if it is possible that any part of g is executed after any part of f in some execution of the program. Similarly, $(f, g) \in SEB$ if and only if it is possible that any part of g is executed before any part of f. The two relations are inverses of each other, so $(f, g) \in SEA \Leftrightarrow (g, f) \in SEB$.

SEA/SEB relations involving (f, g) can be formally defined as follows:

$$SEA = CALL \cup RET \cup SEQ[\cup ID]$$

where $(f, g) \in CALL \Leftrightarrow f$ (transitively) calls g, $(f, g) \in RET \Leftrightarrow f$ (transitively) returns into g, $(f, g) \in SEQ \Leftrightarrow \exists h$: h (transitively) calls f followed by g, and the second call-site is flow-reachable from the first one. Finally, $(f, g) \in ID \Leftrightarrow f = g$. SEB can be formally defined as the inverse of SEA.

2) SEA/SEB for procedures: The reason why SEA and SEB describe safer relations between procedures compared to simple call relations is due to SEQ relations. Thanks to this set, SEA/SEB will discover those (f, g) relations between procedures where f is called followed by g, but g is not called (not even transitively) by f.

In order to compute SEA/SEB, the traditional call graph is not sufficient since the order of call-sites within a procedure body is required to determine the above-mentioned SEQ set of relations. To compute SEA/SEB, the control flow graph (CFG for short) of the system is required. Once we have the CFG, we can compute the dependencies with the help of a language independent algorithm.

An extended example of the above $CRUD$ example (Figure 4) can be seen in Figure 6 with additional SEA/SEB dependencies.

3) Directed SEA-CRUD relations: We mentioned previously that $CRUD$ relations are not directed. The reason for this is that it makes no sense to distinguish between two procedures reading from the same table. It is the same for procedures when updating, inserting, or deleting data from the same table, but it is slightly different in the case of two procedures where one of them modifies the table and the other reads data from the same table. Simply by using $CRUD$ relations for procedures over tables, we cannot determine the execution order of the procedures, so it is not possible to determine whether the procedure reading data from the table reads it after or before the other procedure modifying the same table.

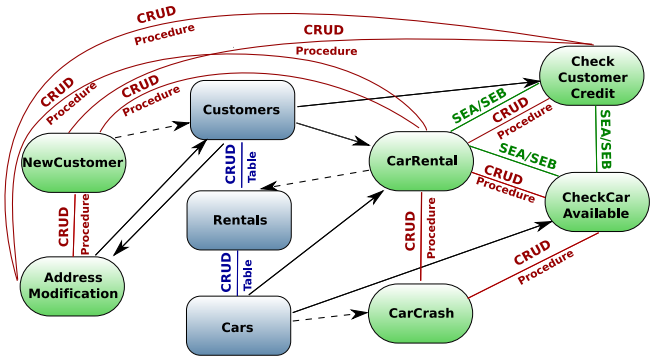

Figure 6. Typical $CRUD$ and SEA/SEB relations between procedures and between tables.

By combining SEA and $CRUD$ it becomes possible to compute which procedure accessed the same table before the other one. We combine SEA and $CRUD$ relations in a simple way; by computing different set operations on the two relations. This way we can use the union as a combined conservative approach, while the intersection as a way to see the stronger relations, for example. This information can be used to evaluate special relations between procedures otherwise not identified.

D. SQL extraction

Although the extraction of SQL commands is not part of the dependency analysis algorithm itself, it has a great influence on the effectiveness of the two methods proposed here. In general, if we can achieve a better extraction of SQL commands, the algorithms will be more precise and more effective.

In many programming languages the SQL queries are sent to the database server by specific procedures which take the SQL commands as a string parameter. These procedures are internal procedures or library procedures accessible via an API. For these languages it is common programming practice to prepare SQL commands by simple string concatenations and to execute the prepared SQL string by giving it as a parameter to one of the internal or API procedures. It has been already shown that for these languages the mere examination of the source code should provide enough information to determine the most significant fragments of the SQL queries. These fragments are sufficient to parse embedded SQL statements and determine the relations using them via a careful static analysis [3].

However, the code fragments of the SQL query may be defined at a certain distance from the place they are used in the source code. In special cases this may result in a situation where determining the exact syntax of SQL commands is unfeasible via static analysis. For instance, when the SQL statement which will be executed is a concatenation of strings where one or more strings are read from the standard input (Figure 7). In this case the executed SQL instruction

```
name = readString();
sql = "SELECT firstname, lastname " +
   "FROM customers " +
   "WHERE firstname " +
   "LIKE('%" + name + "%')";
executeQuery(sql);
```

Figure 7. Example of an embedded SQL query. The query string is concatenated on the second line using a variable in the `WHERE` clause of the SQL query.

could only be captured via dynamic analysis techniques, but it would produce results for only a certain set of executions of the application. However, the SQL command coming from the user input will probably not be the same for the different executions of the application. In order to capture all the possible query strings, one execution is not enough and one must execute the application as many times as the user input may vary. This is usually unfeasible for a large and complex system.

We implemented a static approach for extracting and analyzing embedded SQL commands from the source code. We should mention here that the system on which we evaluated the proposed methods was written in a special procedural language. The programming style of this language makes the whole system strongly database dependent and it makes the use of SQL queries very common in the system. The SQL statements to be executed are embedded as strings sent to specific library procedures and their results will be stored in given variables. This method is actually the same as that for procedural languages where embedded queries are sent to the database via libraries like JDBC or ODBC. This makes our method general and suitable for other languages too.

The implemented approach is based on the simple idea of substituting the unrecognized query fragments in a string concatenation with special substrings. For instance, in Figure 7, it is possible to simply replace the `name` variable with a string "`@@name@@`" and the received query string will be a syntactically correct SQL command. With this simple idea we only need to locate the library procedures sending SQL commands to the database in order to perform the string concatenation, and the above-mentioned substitution of variable, procedure name and other source elements. Whenever the constructed string is syntactically correct, it will have the main characteristics of the executed SQL command.

Developers usually like to prepare statements as close to their execution place as possible and they prefer to keep SQL keywords in separate string literals. In most cases, it is possible to substitute the variables with their last defined values within the same control block. In other cases the variable can be replaced with the variable name.

IV. Evaluation

We performed our measurements on the code supplied by one of our industrial partners. The IT architecture of this company is heterogeneous and it is made up of many different technologies, with a central role of a proprietary technology provided by another local software company. Most of the core systems are built upon this technology, which is an integrated administrative and management system made up of modules (subsystems) using Windows-based user interfaces and MS-SQL databases. The modules contain programs, and the programs are aggregates of procedures. The language is procedural, and its syntax is similar to the Pascal programming language. SQL statements are embedded as strings sent to specific library procedures.

In previous projects, we implemented a source code analyzer for this language (including an analyzer for the embedded SQL language), and many different supporting tools (some of which are language independent). We implemented our methods in this environment and applied them on the working module of a core system.

Table I
Metrics representing the main characteristics of the system.

Metric name	value
(Logical) Lines	$315,078$
Programs	776
Procedures	$2,936$
Triggered procedures	$41,479$
Embedded SQL statements	$7,434$
Tables	317
Temporary tables	641

In Table I, metrics are presented to highlight some of the characteristics of the system being analyzed. We identified $7,434$ embedded SQL strings (based on the specific SQL library procedure calls) and we successfully analyzed $6,499$ SQL statements, which is 87% of all the embedded SQL strings. Here we differentiate between 'normal' and *triggered* procedures. Triggered procedures are assigned to database schemas, tables, and columns. They are never called directly; instead an automated mechanism calls them at runtime whenever a table or column of the schema is used (e. g. written or read). Note that most of the $41,497$ triggered procedures are empty and including them in the measurements adds only 10% more call edges to the call graph (when these triggered procedures call 'normal' procedures). Hence, in the following evaluation of the proposed methods, we focused on 'normal' procedures.

A. Quantitative analysis

In Table II basic statistical indicators of the identified relations are presented. The relations are: (number of) call

graph edges, SEA/SEB relations, while ${CRUD}^{T}_{PP}$ represents the conservative implementation of $CRUD$ relations between procedures, and $CRUD^{*}$ relations representing the variants where only the certain dependencies are considered (dependencies which arise only because of recognized code fragments). The first column shows the total number of computed dependencies for each relation type. The second column shows the maximum number of dependencies of a procedure, and the last two columns are the average and the deviation of dependencies per procedure.

The results in Table II show that there are many relations between procedures via table access which cannot be found using a call graph only. Furthermore, the differences between ${CRUD^{*}}^{T}_{PP}$ and ${CRUD}^{T}_{PP}$ show that the price of a conservative analysis can be quite high.

The average values show that when taking into account the SEA/SEB relations, a procedure may depend on about 8% of the whole module on average. Similarly, with ${CRUD}^{T}_{PP}$ relations, a procedure might be related to about 6% of the other procedures via database access.

In addition to the different relations between procedures, we also measured relations between procedures and tables. The procedures of the system accessed 1.81% of tables on average and 25 was the highest number of accessed tables by the same procedure. This measurement was performed by taking into account only those relations that were not influenced by unrecognized code fragments of SQL instructions.

Table II
BASIC STATISTICS OF DIFFERENT RELATIONS

	sum	*max.*	*avg.*	*dev.*
$CG\ edges$	$18,595$	764	6.33	23.75
SEA/SEB	$727,303$	$2,347$	247.72	361.86
${CRUD}^{T}_{PP}$	$576,095$	$1,066$	192.22	338.87
${CRUD^{*}}^{T}_{PP}$	$156,527$	615	53.31	120.61
${CRUD^{*}}_{CC}$	$1,024,180$	$2,358$	99.69	203.19
${CRUD^{*}}_{TT}$	$11,817$	330	12.23	24.60

The SEA/SEB and $CRUD$ relations have a different basis. Thus, they are comparable as different sets, and one can check whether any of them contains the other, or they are distinct. In Table III, the difference, the intersection, and the union of the ${CRUD}^{T}_{PP}$ and SEA/SEB relations are given. The columns represent the same statistical indicators which were used in Table II. It shows that $CRUD$ and SEA/SEB are different kinds of relation as they have only a few dependencies in common. This means that in the program the two kinds of dataflow (via the normal control flow and via databases) are well separated. Thus, neither of these two relations seems to be better than the other one; they complement each other. However, the intersection of these two types of relations is also interesting. This will determine those dependencies that arise via database access but are potentially used by the same execution (the same operative task). We think that this combined dependency is stronger than any of its components alone, and it can be used to prioritize procedures, e. g., for testing applications, as it will mark only a small fraction of the original relations.

In Table III, we present data for ${CRUDRW}^{T}_{PP}$ relations. This is a special type of the combined relations discussed in Section III-C3, where a read operation in a procedure is followed by a write in another one, or a write followed by a read. ${CRUDRW}^{T}_{PP} \cap SEA$ approximates the database-based dataflow relation of the program. We measured this kind of relation without taking into account those relations which were influenced by unrecognized code fragments of SQL instructions. We found that the rougher relations (${CRUD}^{T}_{PP}$, SEA/SEB), and their combination contained 20% to 70% more edges than the finer ones (with ${CRUDRW}^{T}_{PP}$, SEA).

B. Qualitative analysis

As a qualitative analysis, we manually inspected the computed relations by selecting random samples. We targeted special types of dependencies (e. g. ${CRUD}^{T}_{PP} \cap SEA$, which describes dataflow between two procedures) and we inspected the source code to see whether the chosen dependency actually described a real dependency between the two items.

Most of the evaluated ${CRUD}^{T}_{PP} \cap SEA$ relations were real dependencies between procedures. In some cases, we found that the developers used temporary tables to pass data from one procedure to another one. It is common practice in table manipulation to select data from one table, place it into a temporary table, and later insert the retrieved data from a temporary into different, persistent one. These relations can be readily seen when the procedures working with the same temporary tables are inside the same program.[2] However, it may also happen that procedures in different programs and in different source files of the system have these types of dependencies. An example of a ${CRUD}^{T}_{PP} \cap SEA$ dependency between procedures via a temporary table can be seen in Figure 8.

Another example is about the implementation of *menus* in this framework, which are intensively used so as to let the user access different features. One menu entry executes one procedure of the system. During manual inspection, we found procedures which were in ${CRUD}^{T}_{PP}$ relations, but they implemented functionalities of different menu entries. This means that f and g procedures are in ${CRUD}^{T}_{PP}$ relations, and f is transitively called from the M_1 menu entry while g is transitively called from the M_2 menu entry, but f is not called (transitively) from the M_2 entry, and g is not called (transitively) from the M_1 entry. Finding the

[2]In this framework, *program* is a higher level compilation unit. It contains procedures and it is usually in a separate source file.

Table III
DIFFERENCE, INTERSECTION, AND UNION OF SEA/SEB AND CRUD RELATIONS BETWEEN PROCEDURES

Relations			*sum*	*max.*	*avg.*	*dev.*
$CRUD^T_{PP}$	$\setminus$	SEA/SEB	$542,078$	$1,065$	184.63	97.75
SEA/SEB	$\setminus$	$CRUD^T_{PP}$	$691,782$	$2,346$	235.62	107.54
$CRUD^T_{PP}$	$\cap$	SEA/SEB	$36,037$	369	12.27	10.00
$CRUD^T_{PP}$	$\cup$	SEA/SEB	$1,267,361$	$2,347$	431.66	162.41
$CRUDRW^T_{PP}$	$\cap$	SEA	$29,532$	342	10.06	32.08

relations between these procedures is especially important in this large system where there are around 200 menus working with $2,936$ procedures.

```
procedure procA:
  sql = "SELECT DISTINCT * " +
     "INTO #temptable " +
     "FROM table WHERE condition";
  executeQuery(sql);

procedure procB:
  sql = "INSERT INTO table2" +
     "SELECT * FROM #temptable";
  executeQuery(sql);

procedure procC:
  procA();
  procB();
```

Figure 8. Example use of a temporary table to pass data from one procedure to another one. There is close connection between `procA` and `procB`, as they are in $CRUD^T_{PP} \cap SEA$ relation.

C. Potential applications

Based on the information extracted from the source, we provided support to the company in different areas including software architecture management and software testing. Here, we overview our experiences related to these two potential applications of the dependencies computed with our methods.

1) Architecture reconstruction: In architecture reconstruction, static analysis is used to automatically detect the various relations between software components. With our industrial partner, we previously performed such an analysis between programs using a call graph. We extended this architecture graph using information obtained from $CRUD$ and SEA/SEB. The two relations added $49,754$ additional edges to the $2,459$ original ones between the 584 programs included in the architecture graph. We were also able to include 785 tables and $1,921$ relations between tables and programs in the graph.

2) Regression test selection: Regression tests are carried out to ensure that modifications in the code do not introduce new bugs. However, regression test suites are usually very large, and executing all the test cases for small modifications – despite being safer – can be very expensive resource-wise. Thus, regression test case selection is important. However, executing only those test cases that directly cover the modifications is not always enough, because the change might have an impact in other areas of the system. Impact analysis can be done based on many kinds of dependencies including the call graph, SEA/SEB and $CRUD$.

We computed code coverage (for the change and the corresponding test selection result) using the test execution data of a real testing project. Figure 9 shows how the average coverage value varies with the different impact sets. As can be seen, inspecting more procedures results in a lower coverage. The relation between these two values seems to be linear in our case, but the call graph-based firewall impact results in a smaller coverage value, which was surprising to us. As for SEA/SEB vs. $CRUD$, it can be seen that using SEA/SEB, larger impact sets are obtained and this naturally results in a lower coverage.

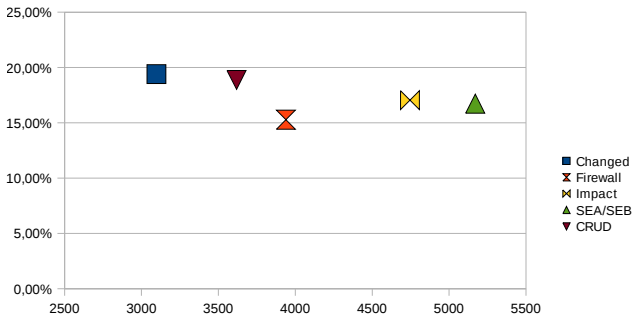

Figure 9. Coverage of different procedure sets. On the X axis the size of the procedure set (corresponding to the different impact analyses) is shown, with the Y axis denoting the coverage values. *Changed* denotes the procedures containing the modifications only; and the other four sets denote procedures that are accessible from Changed procedures via some relations. *Firewall* and *Impact* use the call relation, with only directly accessible procedures, and all procedures that can be reached through a series of call edges, respectively. *SEA/SEB* and *CRUD* denote those procedures that can be reached by traversing the edges of SEA/SEB and $CRUD^T_{PP}$ relations.

D. Implementation pitfalls

In this section we overview some possible threats that might be encountered when implementing the proposed

methods. We encountered these issues as well, so they may serve as "threats to validity" of the results presented above, especially regarding the safety of the analysis.

1) Imprecise ASG: The first main step, which may be unsafe in a complex analysis system, is the source code analysis itself. In this step we extract the AST (Abstract Syntax Tree) from the source code and compute the ASG (Abstract Semantic Graph). In some cases, it may not be possible to build a proper ASG via static analysis. For example, in the languages where dynamic procedure calls are allowed, it is easy to construct source fragments where the called procedure cannot be determined. As the input of our methods, we assume that the input ASG is precise and safe.

2) Unrecognized code fragments in SQL queries: Our SQL extraction method reconstructs the embedded SQL queries with a string substitution rule. We assume that the reconstructed and syntactically correct SQL commands have the same key characteristics as the SQL commands that will be executed by the application. Whenever an SQL query is not parsable, it is handled conservatively and we suppose that it accesses all the tables of the system. However, there are some cases where the SQL query is syntactically correct, but it is not possible to tell which tables it accesses. In Figure 5 we provide an example of this case. This case can be handled by recovering the unknown table, but other problems may arise as well. It may happen that the unrecognized code fragment is in a place of an identifier which is recognized as a column, but it is actually a subquery that accesses several other tables of the database (Figure 10).

```
SELECT firstname, lastname
  FROM customers
  WHERE firstname IN (@@subquery@@);
SELECT @@subquery@@, lastname
  FROM customers;
```

Figure 10. Example of a constructed SQL command where the unrecognized code fragment is in the place of a value, but it is actually a subquery.

The potential error-prone places of the unrecognized code fragment can be determined by a simple rule which states that whenever an unrecognized code fragment is located at a place where it may refer to a subquery or table, then it is assumed that it accesses all the tables of the database.

In later steps we assume that the SQL analyzer and the algorithm which constructs the Usage Matrix are able to recognize all these error-prone cases.

3) Database modifications during code execution: It may happen that while the application is running, the database is modified. If the executed SQL command that produces the change in the database is embedded in the source code, it can be located, but it is hard to tell its influence on the other commands. It may still happen that the database is modified outside the scope of a source code analysis. Our system recognizes the database modification SQL statements, but it does not evaluate them individually. Therefore, they are handled like every other kind of table access.

4) Dependencies via stored procedures: In database-intensive systems it is normal to use stored procedures. Stored procedures are declared and they run on the database side of the application, but it is possible to create and execute them from the application by embedding specific SQL statements (e.g. CREATE PROCEDURE, EXEC). It is important to note that a stored procedure can access database tables like any other SQL command, hence if a procedure of the application executes a stored procedure its Usage Matrix should be properly updated with the accessed tables.

5) Dependencies via internal database dependencies (e.g. triggers, foreign keys): Some dependencies may arise via internal database structures like triggers or foreign keys. These dependencies may lead to a situation where the database manager updates a table due to some modification made in another table. These dependencies can be handled by using an accurate database scheme analysis.

6) Dependencies through temporary tables and views: It is also common in relational database systems to use views for queries or temporary tables to store temporary data. Both of them are sources of hidden dependencies similar to internal database dependencies. Views – like structures selecting data from other tables – can be handled like any other table of the database, but their columns must point to the columns of the original table columns. In the case of temporary tables it is important to bear in mind that it is very hard to follow the lifecycle of a temporary table via a static analysis. Our system currently handles temporary tables like any other table of the system. If one temporary table is created only once inside a compilation unit, all of its references will be properly identified. However if there are other temporary tables created with the same name, it is impossible to determine statically which one is used in a query string.

V. Conclusions

Determining program dependencies in the right way via code analysis is a difficult task. Although many kinds of dependencies and the corresponding methods of analysis have been presented in the literature, they are usually not safe, precise, and efficient at the same time.

In this paper, we presented two methods for recovering program dependencies in database-intensive applications, whose combination is safe in certain situations. One is based on the SEA/SEB relations, and the other uses $CRUD$-based Usage Matrices. We think that the use of these two methods for recovering program dependencies is a novelty here. We performed measurements with experimental implementations of the methods in an industrial context, and presented preliminary results that contain a quantitative

and qualitative comparison of the methods, and also some potential applications.

The results show that the disjoint parts of the relation sets of the two methods are similar in size, and that their intersection is considerably smaller (it is about 3% of the union). So, based on this empirical evaluation, our main conclusion is that neither of the relations is definitely better (safer and more precise) than the other; they are simply different. Thus they should be applied together in situations where a safe result is sought. However, as the corresponding dependency sets are usually different, their intersection could also be interesting in some other situations, such as when a prioritization of the dependencies is necessary, in which case the intersection can act as a higher priority dependency set. However, further research is needed in this respect.

There are many other open questions that deserve to be investigated further. We would like to perform a more thorough comparison of the two methods, where we would take into account some other kinds of dependency analyses published in the literature. As a dependency analysis has many potential application areas, the methods should be evaluated against a variety of applications to see which method is the best choice in a particular case. Finally, the precision of the methods could be improved by applying heuristics that remove certain unnecessary dependencies imposed by the conservative nature of the methods.

ACKNOWLEDGEMENTS

This research was supported by the Hungarian national grants OTKA K-73688 and TECH 08-A2/2-2008-0089.

REFERENCES

[1] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," in *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*. IEEE Computer Society, Oct. 2008, pp. 137–146.

[2] A. Van Deursen and T. Kuipers, "Rapid system understanding: Two COBOL case studies," in *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 1998, p. 90.

[3] H. v. d. Brink, R. v. d. Leek, and J. Visser, "Quality assessment for embedded SQL," in *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 163–170.

[4] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[5] D. Binkley, "Source code analysis: A road map," in *Proceedings of the 2007 Future of Software Engineering (FOSE'07)*. IEEE Computer Society, May 2007, pp. 104–119.

[6] F. Tip, "A survey of program slicing techniques," Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1994.

[7] E. Nyary and H. Sneed, "Software maintenance offloading at the Union Bank of Switzerland," in *Software Maintenance, IEEE International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 1995, p. 98.

[8] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 451–460.

[9] S. K. Gardikiotis and N. Malevris, "A two-folded impact analysis of schema changes on database applications," *International Journal of Automation and Computing*, vol. 6, no. 2, pp. 109–123, 2009.

[10] R. A. Haraty, N. Mansour, and B. Daou, "Regression testing of database applications," in *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2001, pp. 285–289.

[11] R. A. Haraty, N. Mansour, and B. A. Daou, *Advanced Topics in Database Research*. Idea Group Inc, 2004, vol. 3, ch. Regression test selection for database applications.

[12] A. Cleve, J. Henrard, and J.-L. Hainaut, "Data reverse engineering using system dependency graphs," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 157–166.

[13] J. Henrard and J.-L. Hainaut, "Data dependency elicitation in database reverse engineering," in *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2001, p. 11.

[14] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 645–654.

[15] A. S. Christensen, A. Müller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *In Proc. 10th International Static Analysis Symposium, SAS'03, volume 2694 of LNCS*. Springer-Verlag, 2003, pp. 1–18.

[16] J.-L. Hainaut and A. Cleve, "Dynamic analysis of SQL statements in data-intensive programs," University of Namur, Tech. Rep., 2008.

[17] A. Cleve and J.-L. Hainaut, "Dynamic analysis of SQL statements for data-intensive applications reverse engineering," in *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 192–196.

[18] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[19] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile parsing in TXL," *Automated Software Engineering*, vol. 10, no. 4, pp. 311–336, Oct. 2003.