Automated Extraction of Data Lifecycle Support from Database Applications

Kaiping Liu¹, Hee Beng Kuan Tan¹, Xu Chen², Hongyu Zhang³ and Bindu Madhavi Padmanabhuni¹

¹School of Electrical and Electronic Engineering, Nanyang Technological University
{kpliu, ibktan}@ntu.edu.sg
padm0010@e.ntu.edu.sg

²Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China, 100190
chenxu@software.ict.ac.cn

³School of Software, Tsinghua University
Beijing, China, 100084
hongyu@tsinghua.edu.cn

Abstract—Database application is one of the most common types of systems. Grounded on the simple concept of data lifecycle—any data in database is created from insertion, used via selection and modification and terminated at deletion—this paper proposes a novel approach to reverse engineer the data lifecycle automatically from the source code of database applications. The extracted information can be used for the selection of open-source database applications for adaptation. It can also be used for maintenance and verification of database applications. A tool has been developed to implement the proposed approach for PHP-based database applications. Case studies have also been conducted to evaluate the use of the proposed approach.

Keywords—data lifecycle; extraction; reverse engineering; maintenance; verification

I. INTRODUCTION

Due to the complexity of software development and absent or incomplete associated documentation through most stages of the whole software development life cycle, recovery of system inner features automatically from program source code is important in system comprehension, verification and maintenance. Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction [1], and is a suitable approach for this purpose.

During the past decades, much effort has been focused on this area. Michael L. Nelson conducted a survey on the reverse engineering and program comprehension in [2]. He presented various approaches to automating reverse engineering including syntactic analysis, graphing methods and execution and testing method. In 2000, H. A. Muller.et al [3] presented a roadmap of reverse engineering building on the program comprehension theories. They discussed the code reverse engineering and explored the spectrum of reverse engineering tools.

Database application is one of the most common types of systems, it is important to provide support to analyze the code of the database applications automatically. The specific characteristic of database applications is data manipulation and the data maintained in the database is always dynamic. The flow of data processing, from insertion to being used to finally deletion, indicates a lifecycle of data in the database, which can be a representative aspect of database applications.

Based on the basic concept of data lifecycle, this paper proposes a novel approach to reverse engineer the data lifecycle automatically from the source code of database applications. This basic but representative information is an easy to use and understand indicator that benefits the selection, maintenance and verification of database applications.

The paper is organized as follows. Section 2 depicts the data lifecycle. Section 3 discusses the proposed approach. Section 4 reports our evaluation. Section 5 presents the related work and Section 6 concludes the paper.

II. DATA LIFECYCLE

Data maintained in a database is usually dynamic. Once the data is created from insertion, the data should be used, updated, and finally removed once it becomes obsolete. This implies that an attribute and table defined in the schema have a lifecycle that starts from being inserted, via being used as and when needed, and optionally being updated in any number of times, to being deleted. We call such characteristics data lifecycle of attribute and table respectively. Fig. 1 depicts the data lifecycle using state transition diagram.

![Data lifecycle of attribute and table](image-url)

Figure 1. Data lifecycle of attribute and table

The property of the data lifecycle implies directly that any database application must include related operations to support the data lifecycle of attribute and table. Any missing of database operation for supporting the data lifecycle is very serious. Programs in database applications provide support to data lifecycle by performing four database operations: INSERT, SELECT, UPDATE and DELETE.
In a database application, most of the attributes and tables should be provided with a complete data lifecycle. That is, there are programs in the application which perform insertion, selection, updating and deletion for these attributes and tables. However, we should note that there are some special cases on updating and deleting the data, e.g., the application of initial public offer of a company stock. The record can’t be updated or cancelled when it is confirmed.

Table 1 shows all the sixteen possible combinations of data lifecycle support provided in a database application with SELECT, INSERT, UPDATE and DELETE operations. We can classify the sixteen possible combinations according to their implications. For example, if insertion, selection, updating and deletion are all provided for an attribute, then the data lifecycle for that attribute is basically complete. If none of these database operations is provided, then basically the attribute or table is just to be reserved for future use. In addition to the details of data lifecycle support types, Table 1 also shows the classification of data lifecycle support provided in a database application.

### III. Extraction of Data Lifecycle Support

In this section, we give detailed description on the extraction of data lifecycle support from the source code of database applications.

Considering different purposes, our approach extracts data lifecycle support of attribute and table respectively. The data lifecycle support of attribute provides detailed information about the lifecycle support of each attribute and its category. The data lifecycle support of table is the combination of data lifecycle of attributes within the table and provides detailed information about the lifecycle support of each table and its category. Next, we describe the extraction process.

In the extraction, for each SQL query, we need to identify: the operation type (SELECT, UPDATE, INSERT or DELETE); attribute names and table names that are involved. In practice, an SQL query is usually formed as a string literal or more often a string variable, and is then passed to some specific query functions, e.g. `mysql_query` in PHP if MySQL is used. Based on our observation, the value of the SQL string variable commonly varies according to some conditions, i.e. the actual value taken by the query function is subject to different program paths chosen during execution. Fig. 2 shows a simple snippet of PHP code as an example for this.

```
<?php

1.  $attr = 'col1';
2.  else
3.      $attr = 'col2, col3';
4.  endif
5.  mysql_query('SELECT ' . $attr . ' FROM table');
6.  ?>
```

![Figure 2. A PHP code snippet](image)

Table I. Types and Classification of Data Lifecycle Support

<table>
<thead>
<tr>
<th>SELECT</th>
<th>INSERT</th>
<th>UPDATE</th>
<th>DELETE</th>
<th>Category</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Undefined</td>
<td>Data is updated, used and deleted without defined</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Undefined</td>
<td>Data is updated and used without deleted and defined</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Undefined</td>
<td>Data is used and deleted without updated and defined</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Undeletable</td>
<td>Data is defined, used and deleted without deleted</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Redundant</td>
<td>Data is defined and deleted without updated and used</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>Undeletable</td>
<td>Data is defined, used and updated without deleted</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Undeletable</td>
<td>Data is defined and updated without updated and deleted</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Undeletable</td>
<td>Data is defined, used and deleted without deleted</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Reserved</td>
<td>Data is specified without defined, used, updates and deleted</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Complete</td>
<td>Data is specified then defined, used, updates and deleted</td>
</tr>
</tbody>
</table>

From Fig 2, it can be observed that the SQL query in line 6 is constructed by concatenation of three strings, the second of which is a variable with different execution values under different conditions. For the extraction, first, we compute the control flow graph and then perform control and data dependency analysis.

### A. Construction of Control Flow Graph and Control Dependence Analysis

All SQL query function call sites in the source code should be identified, and parameters they take can then be evaluated. We compute standard control flow graph (CFG) for each program. Fig. 3 shows the CFG for the code in Fig. 2.

Starting from the entry block, the control flow graph is traversed and each basic block encountered is checked to see whether it contains query function calls. We can locate one such function call in the node 5 in Fig. 3 representing line 6 in the code in Fig. 2. The parameter which contains the SQL query is extracted. For each query, it would be either one single
variable, or the concatenation of several string literals and variables. In Fig. 2, the parameter only includes one variable: $attr. The actual value of the variables vary according to different condition values taken in the predicate nodes before the function, whose different successive paths would assign different values to the variables that are used directly or indirectly by the query function. To include all possible values for the variables for the precision of the extraction process, we need to further adopt data dependence analysis.

B. Data Dependence Analysis

For each basic node in CFG that contains one or more SQL query function calls, we extract paths in the CFG starting from the entry node and passing through the function call node. For example, from the CFG in Fig. 3, we obtain two paths for node 5. One is 1 -> 2 -> 3 -> 5 and the other is 1 -> 2 -> 4 -> 5.

For each such path, data dependence analysis is performed for every variable that is used in the function call. We trace each variable’s direct or indirect definition from the function call node backward along the path, until either 1) a constant assignment is encountered, for example: $attr = ‘col1’; 2) assignment with values that are unable to be determined, such as return value of user input functions. If the string value of one variable can be determined, this variable is replaced by its actual value; otherwise it is replaced by an empty string. All these strings are concatenated in the same order as they are listed in the function call. By this way, the string value of the parameter used in SQL query function is extracted.

In this way, for the above example, from the first path, we generate the SQL query: “SELECT col1 FROM table”; from the second path, we produce “SELECT col2, col3 FROM table”. These queries are put into a set for further analysis.

C. Data Lifecycle Generation

After the possible values of the SQL string variable used in the query function are retrieved, the set of SQL query strings are formed. When we parse each SQL query, the operation type of the query, i.e., either SELECT, UPDATE, INSERT or DELETE, is first determined. Next, the attribute names and table names are obtained from the strings according to the standard SQL grammar specification. For some set of SQL queries, the operation type, attribute names and table names are identical, and the changing part is the values that are used by the attributes, for example, values in the conditions in WHERE clause. In our approach, these different SQL queries are treated as duplicated and we only keep one of them.

For the example in Fig. 2, we finally extract two selection operations, the first one selects attribute “col1” from table “table”; the second one selects attributes “col2” and “col3” from the same table. We count the two operations into the corresponding data lifecycle information for the attributes and the table.

Finally, with these retrieved attribute operations and their occurrence sites, we integrate them together to count operations on each attribute in each table. Based on these, data lifecycle for attributes and tables as described above can be generated accordingly. Fig. 4 shows the above mentioned data lifecycle extraction process.

Algorithm extractDataLifecycle:
Input: source code files of one software system
Output: Data lifecycle information
Begin:
for each source code file F do:
    Generate control flow graph G for F;
    Traverse G and identify query function invocations;
    for each invocation node I do:
        Extract parameter list from I;
        for each variable V do:
            if each path p in G that passes I do
                Perform data dependency analysis for V on P;
                Trace definitions backward along P;
            else
                Replace V with empty string;
        endIf;
    endFor;
endFor;
Generate a SQL query by concatenation of string values of all variable;
Parse the SQL query and record operation type, attribute names and table names;
endFor;
Integrate all extracted information for each attribute of each table;
Generate data lifecycle;
End.

Figure 4. The pseudo-code of data lifecycle extraction

IV. EVALUATION

In this section, we evaluate our proposed approach for the selection, maintenance and verification of database applications with case studies. We start by first giving a description of our prototype tool.

A. Prototype Tool

Many database applications are developed using PHP. Hence, we implemented a prototype tool in Java for PHP applications in order to verify our proposed approach. PHP is the widely used scripting language for developing web application. We used Pixy [4], an open-source PHP analyzer to parse the PHP source code.

The tool analyzes the source code and extracts the data lifecycle support automatically. It works in the following steps: first it scans through the PHP system and finds all the PHP source code files; second, it parses each file to generate corresponding control flow graph, and then identifies the query
execution functions; third, the tool computes data dependency for the variables involved in the query functions, and based on that possible SQL queries are generated, the operation types and table and attribute names involved are determined; finally, according to the information extracted, the data lifecycle support is generated automatically. Though some variables’ values cannot be decided during data dependency analysis, we observed that these variables are mostly used as the values of attributes, and in most cases the operation types, attribute and table names involved in the queries can be resolved from string literals.

Fig. 5 shows a diagram depicting the workflow of the prototype tool.

![Workflow Diagram](image)

**Figure 5. The workflow of the prototype tool**

### B. Case Studies

#### 1) Selection

Due to the lack of documentation, selection of an appropriate open-source database application for use from the huge repository is not an easy task. Our approach extracts easy to use and understand lifecycle support information automatically to assist the users in the selection process.

To aid the selection process, we give summarized information of the system. With this information, one can assess the quality of the whole system directly from the analysis of data lifecycle completeness. Furthermore, with the information of data lifecycle of attributes and tables, the user can look into detailed information of implementation and be clear about the data lifecycle for each attribute and table.

For the selection process, we evaluated our proposed approach on four real-world PHP DB applications from sourceforge.net. They are School Mate (v1.5.4, a PHP/MySQL school administration and management), Open-School (pre-alpha release, a web-based School Management Software), LAMP School (v0.3 beta, an online school register system) and School Admin (v0.3, a web-based software for schools and colleges). The four systems provide similar functionalities, but differ in their completeness and maturity. Our tool is applied to analyze them and discover their data lifecycle support information.

Due to the space limit, summarized information of two systems is shown in Table II. From the experiment, we know that the percentage of complete category is highest for School Mate (45.3%) followed by LAMP School (42.0%) whereas School Admin has the lowest percentage (1.2%) of complete attributes. The percentage of undefined attribute of School Mate is the lowest (21.3%) while that of Open-School’s is the highest (46.0%). Besides, the percentage of the reserved attributes of School Mate (4.3%) is lower than that of the other three (12.4%, 28.0% and 35.3% for Open-school, LAMP School and School Admin respectively). It also has no undeletable attribute in it. Both the Open School and Lamp School have no redundant attributes or tables while School Admin and School Mate have 8.2% and 6.8% redundant attributes respectively. Furthermore, only in School Admin, the percentage of undeletable category is 17.6% while there are no undeletable attributes and tables amongst the other three.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Total #</th>
<th>Total %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>11</td>
<td>73.3</td>
</tr>
<tr>
<td>Redundant</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Non-modifiable</td>
<td>2</td>
<td>13.3</td>
</tr>
<tr>
<td>Undeletable</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Undefined</td>
<td>2</td>
<td>13.3</td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>15</td>
<td>100</td>
</tr>
</tbody>
</table>

#### 2) Maintenance

The data lifecycle support for attributes and tables is also useful in database application maintenance since it reveals the completeness of functions from inside the source code. In this section, we demonstrate how the data lifecycle support can be used in the maintenance process for cases of undefined, non-modifiable, undeletable, redundant and reserved respectively. We use Table III and Table IV for demonstration.

- **Undefined**

If the data lifecycle support information provided for an attribute or table in a database application is classified as undefined, it is certainly not adequate and additional programs must be coded to address this flaw.
To this information, developers can decide either to add additional programs to use it or removal of unnecessary attribute or table. This may lead to the introduction of redundant, a developer should examine whether it is really needed for the application. This may lead to the introduction of additional programs to use it or removal of unnecessary attribute or table.

As can be seen from Table III, the attribute “reasonofrej” in table “admission” is redundant for it is never used. According to this information, developers can decide either to add functions that use this attribute, or remove it and its other operations correspondingly.

- **Reserved**

If the data lifecycle support information provided for an attribute or table in a database application is reserved, a developer can decide whether he still wants to retain the reserved attribute/table.

From Table III, we learnt that attribute “empid” in table “emp” is reserved. Also from Table IV that, tables “assign”, “rght” and “rolerght” are reserved. This is resulted from that there is not any operation on them. Consequently, whether to delete the attribute or tables from the schema depends on the real adaptation and developers. We found that these tables could be retained because the system is not fully developed yet and further enhancement and extension could be made.

As the above cases demonstrated, data lifecycle information reveals overall status and defect or incompleteness of implementation. Problems with the code could thus be located more easily and quickly. Since software maintenance could often be costly and time-consuming, especially when a new developer tries to have a broad picture of the existing system, data lifecycle can thus be a cost-effective facilitation for the maintenance process.

3) **Verification**

Software verification seeks to provide objective evidence that the design outputs of a particular phase of the software development life cycle meet the specification for that phase. When development reaches a stage, e.g. a milestone, developer would like to check the correctness of the specified programs in that stage. We show a simple example to demonstrate the use of data lifecycle in database application verification.

As is shown in Table V, the attribute “idalunno” in table “alunni” is undefined, i.e. there is no insertion operation for it. However, in the source code file “alu_conf.php”, we found the following SQL query:

"SELECT * FROM alunni WHERE idalunno=$sce"

It can be seen that the attribute “idalunno” in table “alunni” is used as the condition in WHERE clause. However, since its value has never been defined, the result of this selection would be always empty and thus this operation would be inappropriate. This indicates design flaws or incompleteness of implementation against the original design.

Besides, there is another SQL query in a different source code file “del_cat.php” shows the same flaw:

"DELETE FROM cattedre WHERE (idalunno=$_GET[idcl])"

The cases above are only two similar flaws example among all in the source code files. Data lifecycle support helps expose...
such flaws with straightforward information. Reviews can thus be conducted to draw these problems.

V. RELATED WORK

Software reverse engineering plays a critical important role in many aspects such as software maintenance and software comprehension. Over the past decades, a large number of studies have been conducted and quite a number of tools have been developed. In [1], Elliot J. Chikofsky and James H. Cross II et al provided taxonomy of software reverse engineering, in which six definitions of terms were given. Bellay and Gall reported an evaluation of four reverse engineering tools that analyze C source code [5]. They used a number of assessment criteria derived from Brown and Wallnau’s Technology Delta Framework [6].

Reverse engineering has long been employed in software maintenance process. Rainer Koschke surveyed software visualization in software maintenance, reverse engineering and re-engineering in [7], where several representations for both software and specification were described. H. A. Müller presented using reverse engineering approach to aid subsystem identification, which could be used to provide better understanding and maintenance of a large software system [8]. P. Benedusi et al showed the use of hierarchical data flow diagrams in reverse engineering for software maintenance [9]. They focused on methodology at different levels of abstraction. Roger H. L. Chiang et al proposed extraction of ERR model from a relational database through data schema and data instance analysis [10]. However, we focus on extraction of data lifecycle from source code analysis.

Reverse engineering can also be used to enhance the understanding of software systems, and better understanding can result in better maintenance. S. Rugaber gave introductions of methodology, representation and tools in this area of research [11]. In [12], the authors described the results of the use of Rigi project in reverse engineering, which builds mental models from the discovered abstractions. Eleni Strouilia employed dynamic behavior analysis in reverse engineering to understand the system’s process and uses [13]. Instead, our approach uses static analysis. J. Henrand et al described techniques in database reverse engineering [14]. However, they focused on generic methodology for construction of representation from different patterns, while our approach focuses on data lifecycle usages.

VI. CONCLUSION

Though the proposed approach is simple and useful, to the best of our knowledge, no such approach has been proposed in the literature. For a database application, there are four database operations supporting the data lifecycle: INSERT, UPDATE, SELECT and DELETE. We classify the possible sixteen combinations from these four operation types into six categories. We propose an approach to reverse engineer the data lifecycle automatically from the source code of the database applications. To verify the proposed approach, we developed a tool for PHP database applications and conducted case studies to evaluate the proposed approach using the tool.

The first benefit of our approach is the automatic data lifecycle extraction can lighten the burden of analyzing the code manually when the users try to understand the systems’ database usage and behavior. Users can infer design flaws and completeness of the database from the extracted data lifecycle support directly. Secondly, the straightforward information provided by our approach can facilitate the selection, maintenance and verification of database applications.

ACKNOWLEDGMENT

We would like to take this opportunity to express our thanks to the developers of Pixy for releasing their tool as open source software, which helps a lot in our prototype tool development.

REFERENCES