Detection of faults in database applications using static program analysis and data mining

Liu, Kaiping

2016

Liu. K. (2016). Detection of faults in database applications using static program analysis and data mining. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/66667>

<https://doi.org/10.32657/10356/66667>

**DETECTION OF FAULTS IN DATABASE
APPLICATIONS USING STATIC PROGRAM ANALYSIS
AND DATA MINING**

LIU KAIPING

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

2015

**DETECTION OF FAULTS IN DATABASE
APPLICATIONS USING STATIC PROGRAM ANALYSIS
AND DATA MINING**

LIU KAIPING

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2015

ACKNOWLEDGMENT

I would like to express my gratitude to various people who made my life in NTU a fruitful and satisfying journey. Indeed, my PhD life would be scarcely as exciting or insightful as it is without the support of so many friends, colleagues and family members. I am therefore thankful to the following people who shared these memorable times with me.

First, I would like to thank my supervisor, Prof. Tan Hee Beng Kuan, who has provided me with valuable guidance, continuous support and encouragement. Dr. Tan dedicated much time and effort to point me to new research directions and provided various support to me in my moments of difficulty. It is his insistence on excellence that shaped and fine-tuned my research techniques and critical thinking demanded by both the industry and the academia.

My special thanks go to my friend and collaborator, Chen Xu, who contributed greatly to my research work. I would also like to thank my colleagues in Computer Security Lab for their support and help various ways. I would also like to thank Dr. Zhang Hong Yu of Tsinghua University for his kind support in our collaborations together.

Lastly, I thank my parents and Sheldon, to whom I am eternally grateful for their love, care, encouragement, and support over both good and bad times. These are the people that I always turn back to regardless of how things become, and as such they have my special token of gratitude.

TABLE OF CONTENTS

Acknowledgment	v
Table of Contents	vii
Summary	xi
List of Figures	xiv
List of Tables	xv
 Chapter 1 INTRODUCTION	 1
1.1 Motivation	1
1.2 Objectives	2
1.3 Major Contributions	3
1.4 Thesis Organization	5
 Chapter 2 BACKGROUND	 7
2.1 Database Constraints	7
2.1.1 Entity and Referential Constraints	7
2.1.2 DBMS Constraints Enforcement	8
2.1.3 Exception Handling for Key and Referential Constraints	8
2.2 Database Attribute Lifecycle	9
2.3 Program Analysis	10
2.3.1 Static Program Analysis	10
2.3.2 Dynamic Program Analysis	12
2.3.3 Program Analysis for Database Applications	13
2.4 Data Mining	14
2.4.1 Data Preprocessing	14
2.4.2 Data Analysis	15
2.4.3 Anomaly Detection	16
 Chapter 3 AUTOMATED INSERTION OF EXCEPTION HANDLING FOR KEY AND REFERENTIAL CONSTRAINTS	 18
3.1 Database Abstract Syntax Tree Construction	20
3.1.1 SQL Query Extraction	21
3.1.2 Database Information Abstraction	23
3.1.3 Database Abstract Syntax Tree	23

3.2	Automated Exception Handling.....	24
3.2.1	SQL Query Pattern Identification	25
3.2.2	Automated Insertion of Exception Handling	25
3.3	Evaluation	30
3.3.1	A Prototype Tool.....	30
3.3.2	Testing Subjects	31
3.3.3	Results and Discussion.....	31
3.4	Related Work	33
3.5	Conclusion	35
Chapter 4 AIDING MAINTENANCE OF DATABASE APPLICATIONS THROUGH EXTRACTING ATTRIBUTE DEPENDENCY GRAPH		36
4.1	Extraction of Database Attribute Dependency Graph.....	37
4.1.1	Defining Attribute Dependency Graph	38
4.1.2	Control and Data Dependency Analysis	39
4.1.3	Attribute Dependency Analysis	41
4.1.4	Construction of the Graph.....	42
4.2	Prototype Tool.....	42
4.3	Case Studies	43
4.3.1	Testing Dataset.....	43
4.3.2	Dropping Attribute	44
4.3.3	Changing Attribute Definition	46
4.3.4	Changing Program	48
4.3.5	Discussions.....	49
4.4	Related Work	50
4.5	Conclusion	51
Chapter 5 DETECTING ANOMALY IN THE USAGE OF DATABASE ATTRIBUTE		53
5.1	Characterizing Database Operations Using Feature Vectors	56
5.1.1	Database Operations.....	56
5.1.2	Extraction of Feature Vectors for Attribute Usage	57
5.2	Proposed Approach.....	60
5.2.1	Clustering Method.....	60
5.2.2	Labeling Clusters	61
5.2.3	Detection of Anomalies.....	61
5.3	Evaluation and Results.....	62
5.3.1	Training Dataset	62

5.3.2	Performance Measurement.....	63
5.3.3	Fixing Parameters	64
5.3.4	Cross Validation Testing.....	66
5.3.5	Some Examples.....	67
5.4	Discussions.....	68
5.4.1	Variations to Clustering and Detection	68
5.4.2	Assessing Performance	69
5.4.3	Threats to Validity.....	69
5.5	Related Work	69
5.6	Conclusion	71
Chapter 6 ATTRIBUTE LIFECYCLE COVERAGE TESTING OF DATABASE APPLICATIONS.....		73
6.1	Attribute Lifecycle Coverage Testing	74
6.2	Extraction of Conditions of database applications.....	76
6.2.1	SQL Extraction	76
6.2.2	Extraction of Conditions of Database Applications.....	77
6.2.3	Program condition expression extraction.....	78
6.3	Test Coverage Analysis	79
6.4	Evaluation	79
6.4.1	Dataset Description.....	79
6.4.2	Research Questions.....	80
6.4.3	Evaluation Results.....	80
6.4.4	Case study	83
6.5	Related Work	83
6.6	Conclusion	84
Chapter 7 CONCLUSION AND RECOMMENDATIONS.....		85
7.1	Conclusion	85
7.2	Recommendations.....	89
Appendix A.....		92
Author's Publications.....		93
Bibliography		95

SUMMARY

This thesis presents approaches for detecting faults such as violations of constraints in databases and anomaly in the usage of database attributes of database applications. It also introduces a testing method of generating test cases to detect faults of database applications.

The design purpose of Database Management Systems is to solve complicated requirements emerged for offering data consistence as well as persistence. Application examples include concurrent data accessing, complicated transactions execution as well as data analysis upon datasets of large sizes. Most of the up-to-date business systems depend on such Database Management Systems of generality. Hence it is critical to guarantee the correctness of database operations of practical database applications.

Current approaches to detect faults of database applications suffer from some major drawbacks that lead to frequent occurrences of faults. Most of the approaches do not take the interaction between program source code and the database structured query language queries into consideration. Such interaction is actually the key component of database applications because it constitutes the most of the business logic. Furthermore, current methods regarding the impact analysis of database applications either only consider the individual database queries or analyze the program flow dependencies. However, data flow between programs and the Database Management Systems also plays an important role on impact analysis of database applications.

In addition, classification techniques are used for anomaly detection for a long time, but the quantity of training data required for effective classification is typically large. Manual creation of these training data is time-consuming and tedious. Moreover, since the requirement specification keeps evolving, behaviors which are currently normal might become abnormal in the future. For these reasons, classification-based fault detection algorithms, which rely on labeled data, are often inaccurate and highly expensive. Last but not least, rather than using single declarative language, database applications usually are composed by a variety of both imperative and declarative languages. Therefore, existing testing approaches of software systems which are used for imperative languages cannot be applied directly to database applications.

Hence, it is clear that alternative solutions, which are easy to use and yet effective, are required to comprehensively analyze database applications to find faults. Based on the above

motivations, in this thesis, we propose four novel approaches based on prominent static program analysis and data mining techniques.

In relational database, key and referential constraints are key components to ensure accuracy and consistency of data in database management system. Most Database Management Systems automatically enforce key and referential constraints and decline any operations which would lead to constraints violation. However, exception handling for such rejections still requires extra coding efforts by programmers. Current research mainly focuses on maintaining the enforcement of constraints of databases. No research has explored the automatic exception handling for the violations of database constraints. We propose an approach to automatically generate and insert the exception handling code for structured query language queries for the source code in need. This helps to improve programming efficiency and also aids in avoiding coding errors from exception handling and preventing neglect or inconsistent action for handling the same category of exceptions.

As database applications are becoming more and more complicated, this rising complexity calls for more frequent updating in these applications. However, little work has been done in the field of software maintenance research targeting specifically on database applications. Existing approaches do not provide comprehensive information on the dependencies involved in the structured query language queries, so maintainers still have to manually inspect large chunks of code to analyze the impact whenever there is a change. To complement existing approaches, a novel graph structure called the attribute dependency graph is proposed to unveil the intricacy between database attributes and the involved source code. This approach relies on conventional inter-procedural static program analysis to extract the attribute dependency graph.

We propose a clustering-based anomaly detection approach to detect anomaly in the usage of database attributes. We abstract and characterize database operations performed on a database attribute by a feature vector extracted from code through static program analysis. We propose a method to separate database attributes into different clusters by applying a distance-based metric. When the clusters of database attributes have formed, small clusters are identified and labeled as anomalous with the assumption that abnormal attribute would have high chance to be outlier of the dataset due to its rarity. Then the obtained cluster model can be used to detect anomalies from unseen database applications. Our anomaly detection model provides an alternative solution to existing fault detection approaches of database application.

To address the problem that traditional testing method may not be applicable for database applications, we propose a testing approach for the coverage of attribute lifecycles. We also propose a test coverage analysis to measure the quality of a test suite.

This thesis also presents experimental evaluation of each proposed approach and demonstrates that the approaches are useful and effective.

LIST OF FIGURES

Figure 2-1. Database attribute lifecycle	9
Figure 2-2. An example of control flow graph	11
Figure 3-1. An AST example of PHP code.....	21
Figure 3-2. CFG for the program in Figure 3-1(a).....	22
Figure 3-3. A sub-tree of DB-AST	24
Figure 3-4. DB-AST Sub-tree for exception handling	27
Figure 3-5. DB-AST sub-tree for exception prevention	29
Figure 3-6. The work flow of the prototype system.....	30
Figure 3-7. Histograms for the statistics of the four systems.....	32
Figure 4-1. Basic notations of attribute dependency graph.....	38
Figure 4-2. Attribute dependency graph	39
Figure 4-3. PHP code snippets - test.php.....	40
Figure 4-4. CFG for the code of test.php	41
Figure 4-5. Attribute dependency graph for program in Figure 4-3	42
Figure 4-6. The workflow of the prototype tool	43
Figure 4-7. Attribute dependency graph of ECSshop.....	45
Figure 4-8. Attribute dependency graph for FrontAccounting	47
Figure 4-9. Attribute dependency graph for the student league membership management system	49
Figure 5-1. PHP code example	53
Figure 5-2. An example of outliers in a data set with 2 dimensions	55
Figure 5-3. Inter-procedural CFG for PHP code snippet in Figure 5-1	58
Figure 5-4. The algorithm for feature vector extraction.....	60
Figure 5-5. Overview of the proposed approach.....	60
Figure 6-1. Inter-procedural CFG for PHP code snippet	77
Figure 6-2. The ALC and the percentage of path.....	82

LIST OF TABLES

Table 3-1. The experimented PHP database applications	31
Table 3-2. The Exception Handling Requests in PHP database applications detected by GEHPHP	32
Table 4-1. Overview statistics of three systems	44
Table 4-2. Dropped attributes of ECShop.....	45
Table 4-3. A part of changed definitions of attributes of Frontaccounting.....	47
Table 5-1. Systems used in evaluation.....	63
Table 5-2. Statistics for evaluating w.....	64
Table 5-3. Statistics for evaluating p.....	65
Table 5-4. Clustering results of the training dataset.....	66
Table 5-5. Statistics of the three subsets	66
Table 5-6. Performance of the system under various training and testing dataset combinations ...	67
Table 6-1. Statistics of systems in evaluation	80
Table 6-2. Results of evaluation for RQ1	81
Table 6-3. Evaluation results of RQ2.....	81
Table 6-4. ALC results of RQ3	82
Table Appendix-1 Notations used.....	92

Chapter 1

INTRODUCTION

1.1 Motivation

Database management systems (DBMS) are designed to store structured persistent data with formal schemas. DBMS is a crucial component of many software systems and Internet services. For example, banking systems, customer relation management systems as well as enterprise resource management systems all heavily rely on databases. Therefore, it is important to ensure the correctness of the database operations among the systems.

Several problems exist for modern large-scale industrial and open-source database applications. With an expanding business, an initial data model often cannot handle new business logic requirement and specification. Thus database schemas become increasingly complex and error-prone due to cluttering, redundancy and incompleteness. Most importantly, such situations tend to introduce extra overhead of key and referential constraints (KRC) maintenance and exception handling when there is violation of these constraints.

Another notable issue with database applications is that DBMS are all standalone components, which are usually run on separate remote machines and operated by structured query language (SQL) queries. Under this configuration, it is inevitable that the program of the applications contains large amount of interactions against the database by means of queries. However, SQL queries themselves also have conditions and data flow. Historical experience shows that traditional testing methods are difficult to be directly applied to database applications to detect faults.

In practice, database schema is continuously changing because system requirements keep changing in time. Moreover, since the data that is kept within databases is generally dynamic, programs must always be modified to maintain the changeable data. An inadequate change on schema may possibly break the original system. The feature of the dynamic data in database, together with the fact that the interactions between database programs are complex, clearly implies a need for automated tool to detect potential faults.

Like software systems, database applications also have their lifecycles, which may consist of the process of requirement analysis, design, implementation, testing and evolution. Since

every stage of the lifecycles has the possibility of error occurring, it is important to detect faults from multiple stages. Furthermore, database applications have another unique lifecycle, data lifecycle, which consists of the creation, modification and deletion of the data. Since data is the center of the database applications, data lifecycle is of great importance for these systems and would be the focus of this thesis.

Static program analysis [104] is a technique that analyzes the program source code without actually running it. Static program analysis is the prerequisite of many important tasks, including compile optimization, program understanding, automated white-box testing, security auditing, etc. Previously, many researches have been performed on this area. However, as described above, database applications have their own characteristics. Few existing approaches or tools have consideration for these characteristics. By incorporating SQL queries into the static program analysis process, both the data flow and control flow can be extended to reflect the aforementioned database interactions so as to achieve more complex and complete analysis goals. Hence, our challenge here is to propose novel approaches that are practical to aid programmers and maintainers in detecting faults using static program analysis technique.

Data mining [103] is a knowledge discovering process which intends to uncover the underlying patterns via analysis of the statistical characteristics buried in the target sample data. Data mining methods, such as clustering and classification, are also very useful and powerful in the area of software system analysis. For example, in [83], data mining techniques are reviewed to detect malicious programs. [60; 66] explore the possibility to use data mining to detect program error patterns. Data mining can also be applied to extract system specification from code [84]. In this thesis, we attempt to use data mining technique to explore the automatic fault detection for database applications.

1.2 Objectives

The key research objective of this thesis is to discover the applications of static program analysis and data mining techniques in addressing database fault detection issues. Motivated by the above observations, our aim is to propose novel approaches that can be applied to detect and handle faults of database applications in different stages, and to develop tools that realize the proposed approaches.

Database applications have their lifecycles. These stages are internally correlated. For example, maintenance of the database application may request new features to be developed, while new features require additional maintenance. In addition, the testing of the database application usually requires developers to re-examine the program source code to fix the errors. In our research, we focus on the implementation, testing and maintenance stages of the

database applications. At each stage, we design specialized methods to improve and enforce the correctness of database operations as well as the completeness of business functionality. We aim to help the developer by automatically generating the exception detection and handling code regarding the key and referential constraints during the development stage. Furthermore, database attribute is the main part of any database application. We would like to help the developer to detect the missing, redundant or inconsistent operations performed on database attributes. Besides this, database schema keeps changing due to the need of new requirements or functions. This makes database maintenance costly and time consuming. Hence, it is of great importance if there is some effective method to help improve the maintenance efficiency. We aim to provide an approach to aid in impact analysis when the database is changed during maintenance. Last but not least, database application is different from normal programs which use purely imperative languages. We provide a novel testing approach which is suitable for database applications. More specifically, our objective for this research is to address the issues of database applications in the following aspects:

- *Database constraints exception handling*: In this aspect, we aim to identify a few code patterns that require exception handling. We intend to automatically generate and insert exception handling code of KRC violations.
- *Database maintenance*: In this aspect, we aim to provide an approach to effectively help the developer or maintainer to maintain the database consistency when changes happen in the database applications.
- *Database attribute usage anomaly detection*: In this aspect, we aim to extract a set of features from database program to characterize the database attributes. We intend to build effective clustering model from these attributes in order to detect anomalies in the application of database attributes.
- *Database testing*: In this aspect, we aim to provide a testing method which is suitable for database applications. We intend to propose a test case generation method which can be applied to effectively test the faults in database applications.

1.3 Major Contributions

In this thesis, we present four main novel approaches that are intended to detect faults of database applications. These approaches have been cited by a few international journals and prestigious conferences. The major contributions are:

- 1) **Database constraints exception handling**: Most DBMS can automatically enforce KRC by defining them in the database schema, upon which DBMS will prevent any

updating to a database that will cause constraint violation. However, exception handling to handle such rejections still requires coding by programmers. We propose a novel approach to automatically generate and insert exception handling code of KRC violation. Two alternatives are provided for handling the possible exceptions associated with KRC. The first handles the exceptions in conjunction with the built-in enforcement in DBMS, while the second handles the exceptions in program without using the automated enforcement feature provided in DBMS. This work has been published in [57].

- 2) **Database maintenance:** While maintaining database applications, alters made to database operations that are generated by programs commonplace. We propose an attribute dependency graph to assist in determining the impact of such changes with respect to databases which are the most important parts in such systems. With the assistance of our graph, before making a decision to modify or remove the SQL operations in certain programs, developers or maintainers can first refer to the attribute dependency graph of the system and check whether the affected attributes would lead to inconsistency or incompleteness of functionality. The approach can avoid human error in examining and identifying of all the potential impact. Meanwhile it can lighten the burden of the programmer and improve the efficiency of maintenance. This work has been published in [56].
- 3) **Database attribute usage anomaly detection:** In database applications, it is difficult to get labelled data. Clustering techniques are better suited to detect anomalies in database applications because they can be applied to unlabeled data. We present an automatic, clustering-based anomaly detection algorithm, which searches for anomalies in an input set of unlabeled database attributes. We present a method to identify abnormal database attributes by extracting abstract characteristics of database operations executed in database transactions. The proposed approach is capable of inspecting the anomalies of database attribute usage pattern with good accuracy while holding a comparatively small false positive rate. This work has been published in [59].
- 4) **Database testing:** Database applications are typically written in a variety of imperative language and declarative languages, rather than using a purely imperative language. Hence, current testing tools can hardly abide the nature of database applications. However, from our investigation, while conducting database testing, all attribute lifecycle stages in a database application should be exercised. A method to test the coverage of attribute lifecycles has been proposed. This takes into consideration both the program control flow and the conditions in SQL operations. A test coverage analysis to measure the quality of a test suite has also been proposed. The generated

test suites are shown to be able to discover the faults from both program source code and SQL queries.

1.4 Thesis Organization

This thesis is organized as follows.

Chapter 2 provides the background required. It first reviews database constraints, and then presents the database attribute lifecycle. We also discuss static and dynamic program analysis techniques and data mining-based learning schemes, which shall be used in our proposed approaches.

Chapter 3 presents our work on automated insertion of exception handling on KRC. We parse the code of each program using static grammar parser and transform the code into an Abstract Syntax Tree (AST). We then expand the classic AST to accommodate the database operations by taking the database queries into account. We also provide a formal transformation rule which can be applied on the Database-based AST to insert exception handling code. After that, we evaluate the approach by using real-world database applications. A tool called *GEHPHP* has been developed to implement the proposed approach. We check the correctness of the inserted code by triggering the execution of exception handling code.

Chapter 4 focuses on aiding the maintenance of database applications. We propose attribute dependency graph to reveal the dependencies between attributes in a database application and also the programs involved. Furthermore, we propose an approach to automatically extract the attribute dependency graph from the source code of a database application by means of inter-procedural static program analysis. We also present our prototype tool to implement the proposed approach. Case studies have been conducted to look into the impact of changes made to these database applications, including dropping of attributes, change to attributes and changes to programs.

Chapter 5 presents detecting anomalies in database applications through clustering. We differentiate operations of database transactions on database attributes and correspondingly extract feature vectors for database attributes from application source code. After the database attributes are grouped together, the small clusters are identified and labelled as abnormal clusters. To verify our approach, validations are conducted on both open-source and industrial database applications. Finally, results demonstrate that the proposed approach is capable of detecting several types of anomalies with high detection rate while at the same time maintaining a low false positive rate.

Chapter 6 presents a testing method to guide the testing process of database applications. We provide the technique to design test cases from white-box strategies. A test coverage analysis has been proposed to measure the quality of a test suite. We also present our prototype tool that automates the measurement of the coverage of attribute lifecycles.

Finally, the contributions of this research are summarized in Chapter 7 and potential directions for future work have been identified.

Chapter 2

BACKGROUND

This chapter reviews the background of database constraints, database attribute lifecycles, program analysis and data mining techniques. Chapter 2.1 reviews database constraints, including the entity and referential constraints and how these constraints are enforced in database applications. Chapter 2.2 demonstrates the attribute lifecycles. Chapter 2.3 and Chapter 2.4 provide an overview of fundamental program analysis techniques and data mining techniques that we shall apply in our proposed approaches to detect faults of database applications.

2.1 Database Constraints

Data integrity refers to the maintenance of data accuracy and consistency over its entire lifecycle. This is typically enforced in a database system by a series of integrity constraints or rules. Hence, integrity constraints play an important role in database applications. Enforcement of these constraints will reject inconsistent or unauthorized data from entering a database. Three types of integrity constraints are intrinsic to the relational data model: entity integrity, referential integrity and domain integrity. Out of these three types of integrity constraints, entity integrity and referential integrity are the most common integrity constraints which can be automatically enforced by a DBMS once specified in the database schema [85].

2.1.1 Entity and Referential Constraints

An attribute or a set of attributes referring exclusively to each record in a table is determined by the entity constraint. The non-null-valued attribute unique to each individual distinct record is known as the primary key (PK).

The relationship between two tables is established by a referential constraint through the identification of an attribute or a set of attributes in the referencing table that refers to an attribute or a set of attributes in the referenced table. An attribute defined as a foreign key (FK) in the referencing table should be defined as a PK in the corresponding referenced table. If a referencing row's FK has no matching PK value in the referenced table, the referential constraint is violated.

2.1.2 DBMS Constraints Enforcement

A thorough software system should be “defensive” to deal with the KRC violations. There are two basic ways the KRC can be enforced in a database application. The first method is to detect the conditions before a pending SQL operation, and then perform the manipulation if no constraint is violated. The following code snippet shows an example.

```
// check potential key constraint violation
$query = "SELECT * FROM users WHERE username = $username";
$result = mysql_query($query);
if (mysql_num_rows($result) != 0) {
    $message = 'DB query error: ' . mysql_error() . '\n';
    // Show error message and terminate the program
    die($message);
}
$query = "INSERT INTO users VALUES ($username, $password)";
mysql_query($query);
```

In this example, the value assigned to the PK “username” is first checked to see whether it already exists or not. Only when the value does not exist can the new record be inserted into the table.

The other method reverses the enforcement and manipulation phases. The SQL manipulation is first executed to check whether any constraints violation occurs based on the result or the DBMS error code. If there is a KRC violation, all the changes would be deleted and rolled-back. The following code snippet shows an example for this case.

```
$query = "INSERT INTO users VALUES ($username, $password)";
mysql_query($query);
if (mysql_error() != 0) // check whether there is error
{
    $message = 'DB query error: ' . mysql_error() . '\n';
    // Show error message
    die($message);
}
```

Here, the value of the attribute “username” is inserted first, and then the DBMS error code is used to check if any exception occurs. As shown in the above examples, programmers need to perform KRC violation exception handling every time when executing an SQL query. This is obviously cumbersome and error-prone.

2.1.3 Exception Handling for Key and Referential Constraints

Most DBMS can automatically enforce KRC by defining them in the database schema, upon which DBMS will reject any updating to a database that will lead to constraint violation. However, exception handling to handle such rejections still requires coding by programmers. Following is an example which executes an SQL query but without the provision of exception handling.

```
$username = $_GET['username'];
$query = "UPDATE users SET username=$username";
```

In this example, the attribute “username” is a PK. The SQL is executed without any exception handling code following. This may cause key constraint violation. In order to make the program robust and less error-prone, programmers should provide adequate exception handling of KRC violations.

Two alternatives are provided for handling the possible exceptions associated with KRC. The first handles the exceptions in conjunction with the built-in enforcement in DBMS, while the second handles the exceptions in program without using the automated enforcement feature provided in DBMS.

2.2 Database Attribute Lifecycle

In a database application, for each attribute, a value is created initially via insertion. Then, the value can be referenced or updated via selection and updating respectively. Referencing and updating can occur in any order. Eventually, the values of the attributes may be deleted from the database via deletion when they become obsolete. These occurrences of events and states are associated together to constitute attribute lifecycle. Figure 2-1 depicts the state transition diagram of the attribute lifecycles with all the possible events included.

Figure 2-1. Database attribute lifecycle

Looking at the properties of the attribute lifecycle, it is clear that any database application must provide essential events as database operations to support proper attribute lifecycle. Each of these database operations corresponds to a state in the attribute lifecycle. Serious faults and incompleteness of functionalities often imply the lack of these required database operations.

There are programs in database applications that sustain the attribute lifecycle by executing the database operations: INSERT, SELECT, UPDATE and DELETE. Hence, these database operations can be extracted from the code to characterize the attribute lifecycle. For a database attribute, we formulate the following attributes to characterize its lifecycle: Create (C), Null Create (NC), Control Update (COU), Overriding Update (OVU), Cumulating Update (CMU), Delete (D), Use (U), Other Update (OU).

2.3 Program Analysis

Program analysis refers to techniques which automatically examine software programs, either in the form of source code or compiled binaries and extract certain properties in order to serve certain purposes such as performance optimization, security scrutinization and fault detection. Program analysis techniques can be generally divided into two major categories: static program analysis and dynamic program analysis.

2.3.1 Static Program Analysis

Static program analysis techniques analyze the programs without actually executing them. These techniques just parse the program source code files, the object code or program binaries to obtain useful information. The advantages of static program analysis are as follows: 1) it requires no execution environment, which could be inaccessible for the program analysis job; 2) it requires no access to the program input data; 3) the techniques are usually more general to be adapted to different types of programs.

Almost all program analysis process involves control flow analysis and data flow analysis, which is the cornerstone to understand the basic structure and the semantics of a program. Control flow analysis is used to generate control flow dependencies, which are useful to determine the execution flow against all the path conditions. Data flow analysis is built upon control flow analysis to find dependencies between different program variables and can be used to reveal possible execution states under the static analysis context. Control flow and data flow analysis have been applied on several usage cases, for example in white-box testing [46], performance optimization [29], program understanding and bug detection [71]. We use control flow analysis and data flow analysis extensively in our research on database applications to extract useful underlying implementation details and semantics so as to identify potential faults and errors.

We will introduce some basic ideas of the control flow analysis and data flow analysis. The notations and terminologies are mainly from [2].

1) Basic block.

A basic block is the building block of a control flow graph (CFG). It contains a sequence of consecutive statements as in program source code or instructions as in object code, in which flow of control can only enter at the beginning and leave at the end. For conditional control block, the last statement is a conditional jump which can lead to more than one branch.

2) Control flow graph.

A CFG is a directed graph which is constituted with basic blocks as its nodes. The directed edges of the graph denote the control flow of the program. A complete CFG should also have an entry node and an exit node. Figure 2-2 shows an example of CFG, which includes conditional branches and loops.

Figure 2-2. An example of control flow graph

3) Dominator.

If every path from the start node to node b goes through a , node a is said to dominate node b in the CFG. We say that node a is a dominator of node b . A post dominator is just a dominator of the reversed graph of the original CFG.

4) Control node.

If a node on the CFG has more than one (usually two) out-going branches, this node is called a control node. For example, for an “if” statement commonly seen in modern programming languages, there are two branches (“true” and “false”) associated with it on the corresponding CFG.

5) Control dependence.

Node a is control dependent on Node b if and only if b has successors s_1 and s_2 such that a post-dominates s_1 but does not post-dominates s_2 . In general, if a node a is control dependent on a node b , then b must be a predicate node.

6) Definition and use.

Definition and use analysis (def-use) is a basic technique to determine data flow dependencies. For a statement $S: v_1 = v_2 + v_3$, we say that there is a definition for v_1 in this statement, and use for v_2 and v_3 . Def-use analysis can be used to perform reaching analysis, dead code identification, taint tracking, constant propagation, etc.

7) Working list algorithm

The most commonly used algorithm to analyze data flow is the working list algorithm. Basically, this algorithm iterates each basic block to see if it generates more definition after its scope. The iterative algorithm terminates when no new definition is produced.

8) Symbolic execution

Symbolic execution involves an interpreter to parse every program statement or instruction and preserve data flow dependencies by assigning symbolic values, rather than actual running values to each of the variables of the program.

We use the algorithms to calculate CFG, control dependencies and data dependencies provided by Ferrante et al. [29], Lengauer and Tarjan [51], and Korel [46].

2.3.2 Dynamic Program Analysis

Dynamic program analysis is performed while the program is running, either by monitoring its general states as a bystander, or by using a container like a virtual machine to record its detailed behavior [19]. Dynamic program analysis is used in black-box testing [4; 99], binary instrumentation, debugging, profiling [10], etc.

Unlike static program analysis, dynamic program analysis can allow the program subjects to exhibit its actual runtime behavior with real-world input data, which could be a difficult task

for the static program analysis [88]. However, one disadvantage of dynamic program analysis is that it could often impose significant performance penalty on the subjects, especially when the analysis requires highly accurate results.

1) Binary profiling and instrumentation.

Tools like Pin [61] and Valgrind [68] are built to perform dynamic profiling for binary executables. Data collected from the profiling can be used to find out where the performance bottleneck lies. For example, Valgrind can be used to count the function call time costs and monitor the CPU cache miss rate.

2) Test input generation.

Research works like [4; 99] seek to use dynamic program analysis to generate test input suite. These techniques collect program control flow conditional constraints during execution and try to produce input combinations for exercising different program paths using constraint solvers.

3) Security sanitization

By tracking the data flow from user input to program output, dynamic analysis is able to detect security and privacy leak by checking if the data flow breaches some preset rules. Such analysis can be applied to both binary programs as well as web applications. OWASP [7; 72] and RSnake [80; 81] provide useful tools and libraries which could test the business applications to find vulnerabilities. By simulating functions with such attack inputs and analyzing their output behaviors, one could profile functional behaviors and detect anomalies.

2.3.3 Program Analysis for Database Applications

Database applications usually have heavy dependency on the operations of backend database systems and these operations are performed via SQL queries. Thus SQL queries are crucial components of the systems to ensure correctness and security. As such, the analysis of database applications should take SQL queries into consideration.

Static Program analysis for database applications should include SQL queries into control flow and data flow analysis. The queries are from the program source code, SQL batch files, database storage procedures, etc. [17]. Collecting complete and correct queries is also a challenge for modern large scale multi-tier business systems.

For dynamic program analysis, the execution of SQL queries should also be traced in order to achieve the analysis goals such as security scrutinization. However, database systems are

often run in a different process which is probably on a remote machine. This imposes extra difficulty for such program analysis.

2.4 Data Mining

Data mining is a sub area in machine learning and artificial intelligence which has been receiving intensive research interest and been applied to a large variety of application fields. Data mining is a process that aims at revealing hidden, unprecedented information which is potentially valuable and understandable [103]. It is a decision-based process rooted by the interdisciplinary from several research fields including but not limited to artificial intelligence, machine learning, pattern recognition, statistics, database and data visualization [38]. Through highly automatic analysis of enterprise datasets, data mining can be used to provide reasoning inductively, find underlying common business patterns, help decision makers adjust their market strategies and reduce risks. Data mining is often viewed as a computer science field since the whole process requires the help of great computational power.

In software development, there is also large amount of data collected from source code, bug trackers, specifications and documentations. These data can be used to discover potential program errors and faults, incompleteness of business logic, common patterns of issues and bugs. Data mining is a strong tool that could assist these kinds of discovery processes and aid improvement of software quality and productivity.

Data mining usually consists of the following sub processes, including data preprocessing, data analysis and mining, as well as result evaluation. These steps are briefly discussed in the following subchapters.

2.4.1 Data Preprocessing

In real world applications, the collected raw data is normally in disorder and cannot be used to perform data mining tasks directly. Data preprocessing is a necessary prerequisite step before any data analysis can be carried out. The common problems of raw data are that: 1) data may contain noise, which means some part of the gathered data is too abnormal to reflex actual truth; 2) data may be missing/incomplete, inconsistent and even wrong; 3) the amount of data could be too large to be practical for automatic analysis.

To make it feasible for mining valuable information and knowledge from the raw data, data preprocessing is a necessary prerequisite. We will briefly discuss some of the commonly used techniques in this subchapter.

1) Data Cleaning.

As mentioned above, raw data collected from real world scenarios could often be incomplete, dirty and noisy. There are several possible ways to address missing data. For example, human resources can be deployed to fix those empty cells in the database. However, this is not so feasible for large scale of data. In many data mining applications, general statistics are the main focus. Under such circumstances, the following strategies can be applied so as not to affect the statistical characteristics: filling the missing cell with a global constant, filling with global average value, filling with a most commonly seen value.

For dirty or so called noisy data, data smoothing techniques can be used to address the problem. One way of smoothing is to divide the data into several ranges, and replace the noisy point with the average value within each of the range. Another practice is to perform a clustering process on the data in order to separate the noisy data, i.e. outliers from normal data. In later data analysis process, outliers can be ignored. We will discuss clustering techniques shortly.

2) Data Transformation.

The purpose of data transformation is to convert raw data into certain format that is suitable for machine analysis, e.g. some structure designed for a specific data mining algorithm or a standard process. Techniques used for data include:

Smoothing: remove noisy data.

Aggregation: Merge raw data based on some attributes, e.g. average per day, total income per month/year, etc. This step is usually performed before multi-dimensional analysis.

Normalization: Convert data points of different attributes into same scale. Some statistical machine learning algorithms rely on this step for better performance. Common normalization methods are min-max normalization, and z-score normalization.

2.4.2 Data Analysis

Over the years, several data mining algorithms are developed for different purposes of mining jobs. In this subchapter, we will briefly introduce some widely used data mining tasks which are also employed in the research work of this thesis as well as the underlying algorithms.

- 1) Association Rule Mining: This data mining task aims at finding hidden relationship from series of events. A famous application example is the shopping basket analysis which leads to the information about which groceries are often purchased together. More formally, let A and B denote two items, and if $P(A \cup B)$ (support) and $P(B|A)$

(confidence) suffice some preset thresholds, we can say that A and B is an association rule.

The classic algorithm for mining association rules from database is Apriori [1]. It basically contains two steps: first, generate candidate itemsets based on common prefix; second, prune out the itemsets which do not suffice association rule threshold. One problem of the Apriori algorithm is that the candidate sets could be very large and infeasible to generate. Other algorithms, e.g. FP-growth [35], try to mine association rules without such candidate generation step.

- 2) Clustering: Clustering refers to a large category of algorithms which try to separate data points into different groups, and each of such groups is called a cluster. Data points which fall into the same cluster have similar attributes or statistical characteristics, while data points from different clusters have significant differences.

One of the most famous clustering algorithm is K-means algorithm. Basically, k-means select or assign K cluster center of centroid, and assign every data point to the most suitable cluster based on some metrics on the data point itself and the cluster centers. After all data points are assigned, the K centers are updated based on the assignment, and the process is carried on again until the centers are not changed. This clustering process is a simple form of expectation-maximization, which requires no prior knowledge about the data, i.e. the process is unsupervised. Other clustering algorithm, like hierarchical clustering, have different grouping method [38].

- 3) Classification: Classification has a wide spectrum of applications. For example, predicting transaction risk, classify customer propensity, etc. Unlike clustering, classification has a supervised training process, which means labeled data is required for the training algorithms. This also indicates that classification has higher accuracy regarding prediction since it has prior knowledge. On the other hand, classification needs extra man power to generate the labels. Commonly used classification algorithms include decision tree, Bayesian, support vector machine, etc. All these algorithms try to separate the data points based on predefined gain functions in order to make sure the gaps between different classes of data are as large as possible.

2.4.3 Anomaly Detection

In data mining, anomaly detection is to identify objects, events or patterns which do not comply or conform to normal behaviors or distributions with other majorities. Such abnormal entities are referred to as anomaly. Both classification and clustering techniques are commonly used to detect anomalies [15].

- 1) Classification based anomaly detection. A classifier is trained to distinguish between normal and abnormal cases. Such methods require the training data is labeled with classes so that classification training algorithms can be applied. Multi-class classifier assumes that normal data belongs to multiple normal classes. If an instance is evaluated against that trained model and cannot be classified into any of the classes, it is considered an anomaly [22]. One-class classifier assumes that all normal cases belong to only one class, and such algorithms, like one-class SVM [82], can be used to train and test anomalies.
- 2) Clustering based anomaly detection. These techniques are based on the observation that normal instances would naturally form clusters due to their inherent similarities, i.e. they are neighbors. Meanwhile, abnormal instances usually exhibit isolated attributes with no or few neighbors. Hence, this distribution characteristics can be utilized by clustering techniques such as K-nearest neighbor [33], hierarchical clustering approaches [44] and K-means [86] to detect anomalies.

Chapter 3

AUTOMATED INSERTION OF EXCEPTION HANDLING FOR KEY AND REFERENTIAL CONSTRAINTS¹

The effective and efficient maintenance of DBMS constraints enforcement has been the subject of much research in the area of database research [12]. A set of quality metrics to measure referential completeness and consistency had been put forth by Ordonez and García-García [70]. Madiraju et al. [62] proposed integrity constraint checking for multiple XML databases. Lee et al. [50] came up with embedding data integrity into quality management cycle, but focused mostly on DBMS implementation of constraint violation checking. In our previous work [108], code patterns are used to check the enforcement of integrity constraints. There are some interesting work elaborated by [8] which reverse-engineered a vendor's database as part of an evaluation to determine overall software quality prior to buying a product. Berztiss and Thalheim [6] performed classification of the exceptions in information systems, and discussed how to detect and handle them. However, no research has explored the automatic exception handling for the violations of integrity constraints of database applications.

In recent years, static program analysis has been applied on database applications for several purposes. Jovanovic, Kruegel and Kirda developed Pixy, an open-source tool for detecting cross-site scripting vulnerabilities in PHP programs [41], [42]. In [93], the authors proposed generating securely prepared SQL statement to prevent SQL injection attacks. SAFELI, a static analysis framework for identifying SQL injection attack vulnerabilities, had been introduced by Fu and Qian [30]. The SAFELI inspected the bytecode of ASP.NET applications, and used constraint solver for all SQL query summit points in order to determine whether the SQL query is safe. A proposal to use lattice-based static analysis algorithm is put forth by Huang et al. [37] to inspect web applications. The instrumented code would protect any code sections identified as vulnerable. They are mainly concentrated on security issues of database applications. In [101], the authors proposed an inter-procedural query extraction approach to recover explicit database queries for the program to execute. Dasgupta et al. [20]

¹ This chapter is the author's version of the work. It is posted here by permission of Journal of Database Management (JDM). Not for redistribution. The definitive version was published in Liu, Kaiping, Hee Beng Kuan Tan, and Xu Chen. "Automated Insertion of Exception Handling for Key and Referential Constraints" Journal of Database Management (JDM) 24.1 (2013): 1-19.

also provided a static analysis approach to extract SQL queries from ADO.NET binaries. These works mainly emphasized on reconstruction of complete SQL queries that embedded in database applications.

Program transformation refers to the process that accepts a piece of program and transforms it into a new one. This is usually used in source-to-source compiler which generally makes use of AST as an intermediate representation. For example, in [21], source-to-source compiler is applied to automatically transform C code into parallel programs. Similarly in [53], technique is employed to perform program optimization. To simplify the program transformation process, some transforming languages are designed, including Stratego [11] and ANLTR [73; 74]. ANLTR also provides grammar rule to facilitate the tree transformation. Unlike other works, such as [63] which targeted at other applications, our approach focused on automatic SQL query pattern identification and exception handling code insertion by using expanded AST, which can be viewed as program transformation. To the best of our knowledge, there is no related or similar research that uses program transformation to automatically insert database constraint violation exception handling code into the original programs.

The above observations serve as our main motivation for this work. In this chapter, we propose an approach to automatically transform a program which lacks in KRC violation exception handling into one which is equipped with the required exception handling code while preserving the original semantics.

Contributions and Results

- We parse the code of each program using static grammar parser and transform the code into an AST.
- We expand the classic AST to accommodate the database operations by taking the database queries into account.
- We also provide a formal transformation rule which can be applied on the database abstract syntax tree (DB-AST) to insert KRC violation exception handling.
- We evaluate the approach by using four real-world database applications. We check the correctness of the inserted code by triggering the execution of exception handling code.

This chapter is organized as follows. Chapter 3.1 discusses how the AST is constructed. Chapter 3.2 proposes the approach to handle the exception automatically. Chapter 3.3 describes our prototype tool, the datasets collected for experiments, and the evaluation results of the applicability of the approach. Chapter 3.4 presents the related work. Chapter 3.5 concludes this chapter.

3.1 Database Abstract Syntax Tree Construction

AST is a commonly used tree structure in compilers and program analysis as an intermediate representation of the syntactic structure of the original program. Each node on the tree denotes a grammatical element in a program. The expression evaluation order is expressed implicitly by the tree, i.e. there is no need to express parenthesis on the tree. Figure 3-1 shows an example of AST. Figure 3-1(a) shows a snippet of PHP code and Figure 3-1(b) to Figure 3-1(d) show the corresponding ASTs.

```
<?php
.....
$user_name = _GET['usr'];
$password = _GET['pwd'];
if (NOT_ADMIN)
    $table = 'users';
else
    $table = 'admin';
mysql('INSERT INTO ' . $table . ' (userName, password) VALUES
    ' . $user_name . ', ' . $password );
.....
?>
```

(a) A PHP code snippet

(b) AST sub-tree for the assignment statements

(c) AST sub-tree for IF construct

Figure 3-1. An AST example of PHP code

The code snippet in Figure 3-1(a) is a typical database manipulation process used for user registration. It first obtains a user name and password from HTTP session input data, and then inserts a new record to the corresponding table. Figure 3-1(b) shows the sub-tree of the assignment statements. Figure 3-1(c) depicts a typical conditional branch sub-tree, in which there are three child nodes for the IF-Condition node. The sub-tree of the function call “mysql_query” is shown in Figure 3-1(d). In this function call, there is only one string parameter, which is the concatenation of several string literals and variables.

The classic AST can only express the database queries as simple strings. Hence improvements on the AST should be made to accommodate DB operations information, such as operation types, attribute names, table names, PKs and FKs. To achieve this target, we need to identify and extract all the SQL queries from the code, parse the queries and expand the AST to produce DB-AST.

3.1.1 SQL Query Extraction

For database applications, SQL queries are usually coded as parameters of some specific query functions. In order to locate all such function calls in a program, Inter-procedural CFG is computed. The CFG is traversed starting from the entry block in a depth-first manner. Each basic block along the path is checked to see whether it contains query function calls. For each encountered query function call, its parameters related to SQL queries are extracted. We use the example in Figure 3-1(a) for demonstration. For the program in Figure 3-1(a), the corresponding CFG is shown in Figure 3-2.

Figure 3-2. CFG for the program in Figure 3-1(a)

In the CFG, we encounter query function calls in two different paths. The first path is “Entry→1→2→3→4→6→Exit” while the second is “Entry→1→2→3→5→6→Exit”. Parameters which are strings involved in these two query function calls are extracted. The values of the string variables such as the “\$table” cannot be determined directly on the call sites since they would differ in different paths. In order to obtain the affected table names and attribute names of the queries, data dependency analysis is applied for each variable involved in the query functions’ parameters.

Every path is traced backward to uncover the definitions affecting the variables, which are replaced by their actual values if the values can be determined statically. If the value of variables cannot be determined, they are replaced by the variable names while preserving the SQL grammar. In practice, users of an application usually have no knowledge about the database structures, so it is unlikely that table names and attribute names in the queries are given by user input. Therefore the SQL queries can be reconstructed by concatenating the strings following their original order without losing any information we are interested in. For the program in Figure 3-1(a), two SQL queries are extracted: 1) “INSERT INTO users (userName, password) VALUES ‘\$user_name’, ‘\$password’”; 2) “INSERT INTO admin (userName, password) VALUES ‘\$user_name’, ‘\$password’”.

3.1.2 Database Information Abstraction

The extracted SQL queries are parsed by a standard SQL grammar parser to obtain the operation types, attribute names as well as table names. We use the schema definition to build a key and referential relationship map involving each attribute. This map records whether one attribute is a PK or a FK as well as its referential relation. Finally, for all queries, the following information is extracted: operation type, table names, attribute names, PKs, and FKs. Furthermore, for INSERT and UPDATE queries, the values to be assigned to the PKs and FKs must also be known. For DELETE operations, the PKs could be referenced by FKs of other tables, thus we need to record the corresponding FKs for the PKs. For example, for the query “INSERT INTO users (userName, password) VALUES ‘\$user_name’, ‘\$password’”, the following information is extracted:

Op Type: INSERT;

Table Names: users

Attribute Names: username, password;

Primary Key-Value pairs: username-\$user_name;

Foreign Key-Value pairs: None.

With all such information, we shall be able to expand the classic AST.

3.1.3 Database Abstract Syntax Tree

DB-AST is the acronym for Database Query Aware AST. For classic AST, one database query is simply represented as a string node. In order to represent the database query information on an AST, we create tree nodes to represent the above-mentioned five types of information and append these nodes to the AST. Formally, the extended part of the AST can be described using the Extended Backus-Naur Form (EBNF) [102] similar to the grammar used in ANLTR [73]:

```
DB-Query:
    Operation Table+ Referred_table* Attributes;
Operation:
    SELECT |
    INSERT |
    UPDATE |
    DELETE;
Table:
    $table;
Referred_table:
    $referred_table;
Attributes:
    (PK | FK) + | NK;
PK:
    $PK
    $PK_Value;
```

```

FK:
    $FK
    $FK_Value;

```

In the EBNF, the symbols starting with \$ are the actual values in the query. For instance, for query “INSERT INTO users (userName, password) VALUES ‘\$user_name’, ‘\$password’”, the \$table is “users”, the \$PK is “userName”, and the \$PK_Value is ‘\$user_name’. This is a simplification of the original ANTLR grammar. The classic AST shown in Figure 3-1(d) would be expanded as shown in Figure 3-3.

In Figure 3-3, the nodes in light blue are the appended DB query-related nodes which are under the node “DB-Query”, which becomes the first child node of the node “Parameters”. “DB-Query” node has three direct children: “Operation”, “Table” and “Attributes”. Here, “Attributes” may have the children “PK”, “FK” as well as “NK” (None Keys). In Figure 3-3, FK node is omitted since the query does not include it. Such expanded AST is therefore called DB-AST, which can be used to facilitate the program transformation process.

Figure 3-3. A sub-tree of DB-AST

3.2 Automated Exception Handling

With the DB-AST, SQL query patterns that would potentially violate the KRC can be automatically identified. If potential violation is identified, the DB-AST will be transformed by using the transformation rules. Eventually the transformed DB-AST with exception handling will be written back into source code.

3.2.1 SQL Query Pattern Identification

Operations that modify the PKs and FKs would potentially cause violation of KRC. There are five SQL query patterns that make such modifications, leading to KRC violations: INSERT PK, INSERT FK, UPDATE PK, UPDATE FK and DELETE PK [47]:

- INSERT PK: Insertion of a record containing primary key;
- INSERT FK: Insertion of a record containing foreign key;
- UPDATE PK: Update a record containing primary key;
- UPDATE FK: Update a record containing foreign key;
- DELETE PK: Delete a record containing primary key;

Note that deleting a record only containing foreign key would not cause any violation since it does not affect the primary key.

For better understanding these patterns, we express the patterns as follows:

```
INSERT-PK:
    (Operation: INSERT)
    (Attributes: (PK) *);
INSERT-FK:
    (Operation: INSERT)
    (Attributes: (FK) *);
UPDATE-PK:
    (Operation: UPDATE)
    (Attributes: (PK) *);
UPDATE-FK:
    (Operation: UPDATE)
    (Attributes: (FK) *);
DELETE-PK:
    (Operation: DELETE)
    (Attributes: (PK) *);
PK:
    $PK
    $PK_Value;
FK:
    $FK
    $FK_Value;
```

3.2.2 Automated Insertion of Exception Handling

After the queries that match potential KRC violation patterns are identified, tree transformation is conducted to insert the exception handling code into the DB-AST. For the first kind of exception handling which assumes that the DBMS has already enabled the exception detection, we only need to insert the post-condition that checks the return value of the query execution function. The transformation can be expressed as an EBNF rule of the form “DB_AST $\rightarrow$ Transformed_AST”. The exception handling AST sub-tree should be inserted as the next sibling of the SQL query function call node. The exception handling transformation rule is shown as follows:

```

// Exception Handling Rule
Query-Function-Call -> Query-Function-Call-with-Exception-Handling;

Query-Function-Call:
    INSERT-PK |
    INSERT-FK |
    UPDATE-PK |
    UPDATE-FK |
    DELETE-PK;

Query-Function-Call-with-Exception-Handling:
    Query-Function-Call
    Exception-Handling

Exception-Handling:
    Post-condition;

Post-condition:
    IF-Condition Branch1;

IF-Condition:
    Expression:
        !=
        Error-Function-Call
    0;

Error-Function-Call:
    mysql_error;

Branch1:
    ..... // arbitrary error message
    Terminate-Function-Call'

Terminate-Function-Call:
    die;

```

This rule can deal with all five types of KRC violation patterns and perform transformation. On the right hand side of the arrow in the first line, an Exception-Handling AST construct is added after the query. It includes the post-condition for checking DBMS error status by invoking function `mysql_error`. If the error number returned is not 0, which signals errors occurring, the program will be terminated. Various error messages can be defined and displayed for users.

The inserted exception handling code can be customized by users according to business logic. Here we show a simple example of the inserted code as a demonstration that simply shows an error message. The code snippet actually handles the INSERT-PK pattern:

```

$result = mysql_query("INSERT INTO ...");
// Below is the inserted code
if (mysql_error() != 0) // Post Condition: checking errors
{
    $message = 'DB query error: ' . mysql_error() . '\n';
    // Show error message
    die($message);
}

```

The sub-tree of the exception handling code is shown in Figure 3-4.

Figure 3-4. DB-AST Sub-tree for exception handling

The second approach of exception handling is for the case when the DBMS does not enforce the KRC such that the KRC violations have to be detected by programmers. In order to check for potential PK constraint violation, we need to examine whether a value identical to the inserted or updated already exists, and this method can be viewed as exception prevention. In order to identify potential referential constraint violation, the referenced or referencing attribute should be investigated. It can be done by generating a SELECT query to find the corresponding key in the table. Information taken from the original AST including the table, attribute, PK and FK names, and also the variables or constants assigned to the keys are needed. The transformation rule of exception prevention for the five types of patterns is shown below:

```

// Exception Prevention Rule
Query-Function-Call -> Query-Function-Call with Exception-Prevention;
Query-Function-Call:
    INSERT-PK |
    INSERT-FK |
    UPDATE-PK |
    UPDATE-FK |
    DELETE-PK;
Exception-Prevention:
    Precondition;
Precondition:
    Constraint-Query IF-Condition Branch1;
Constraint-Query:
    Query Query-Execution;
Query:
    "SELECT * FROM" ($table|$refered_table) "WHERE" (($PK $PK_Value) | ($FK
$FK_Value));
Query-Execution returns [Return-val]:
    .....
IF-Condition:
    Expression:
        !=

```

```

        Num_of_Rows
        0;
Num_of_Rows:
    mysql_num_rows;
Branch1:
    .....
    Terminate-Function-Call;
Terminate-Function-Call:
    die;

```

In this transformation rule, the Exception-Prevention sub-tree is inserted before the query, which performs the potential KRC violation checking. If the SELECT query result is not empty, there would be a KRC violation. Hence, the original query should be aborted.

For example, let us assume that there is a table “user” which includes a PK “username”. This table would be modified by inserting a new record. PK value is assigned through the variable \$user_name. This is an instance of the INSERT-PK patterns. Before insertion, the value of \$user_name to set the PK “username” should be checked to ensure its uniqueness. Thus, the following query should be executed:

```
SELECT * FROM users WHERE username=$user_name;
```

If the result set is not empty, there would be key constraint violation. Hence, the precondition fails and the INSERT operation should be prevented from execution. The complete constraints violation prevention code is shown as follows.

```

$tmp_query = 'SELECT * FROM users WHERE username=$user_name';
$tmp_result = mysql_query($tmp_query);
if (mysql_num_rows($tmp_result) != 0) // the precondition
{
    $message = 'constraint violation';
    die($message);
}

```

It should be noted again that the content of the error message can be configured to suit different needs. The corresponding sub-tree of DB-AST is shown in Figure 3-5.

Figure 3-5. DB-AST sub-tree for exception prevention

For the INSERT-FK and UPDATE-FK, the differences of the generated code are that the table and attribute names should be the PKs referenced by the FKs. Similarly, for DELETE-PK, the table and attribute names of the referencing FKs should be used to generate the SELECT operation.

Such inserted prevention code would execute extra SQL queries, but they are necessary to maintain correct business logic and are usually more elegant than exception handling. Besides, we carefully choose only primary keys and foreign keys as the selection attributes, which are already indexed by DBMS systems. Operations on such attribute would not incur too much overhead.

By applying the tree transformation rules on the DB-AST, we can produce a new DB-AST with either exception handling or prevention which is required in a program. The new DB-AST except the DB-Query sub-tree will be written back into source code to generate the complete program.

3.3 Evaluation

3.3.1 A Prototype Tool

To verify the proposed approach, we have implemented a tool named GEHPHP (Generation of Exception Handling for PHP Systems) and conducted experiments on real-world PHP systems. Our tool uses phc, an open-source tool for parsing and compiling PHP programs [75] to generate CFGs and perform control and data dependency analysis. Consequently, SQL queries are extracted from the code as described in Chapter 3.2. After the queries' patterns are identified, ANTLR is used to generate the classic ASTs which are further expanded into DB-ASTs. The DB-ASTs are checked to see if they match one of the KRC violation patterns. If the match is successful, transformation rules are applied on the DB-ASTs to produce new DB-ASTs which involve the corresponding exception handling or exception prevention sub-trees. Finally, the new DB-ASTs are written back into source code. The work flow of our tool is shown in Figure 3-6.

Figure 3-6. The work flow of the prototype system

Using GEHPHP, a study is conducted to evaluate our pattern-based approach and examine the correctness of the code automatically inserted to handle KRC violation exceptions. We created a PHP runtime environment for our experiment on a desktop PC. Apache Web Server 2.2.16, PHP 5.2.14 and MySQL 5.0.90 were installed with default options. The KRC checking functions of MySQL were turned on. The databases and schemas that are used in the testing open-source PHP systems were set up in MySQL in advance.

3.3.2 Testing Subjects

We evaluated the approach on four real-world PHP database applications from the open-source web site sourceforge.net. The four applications are OpenDocMan (v1.2.4, a web-based document management system), AFCommerceShoppingCart (v1.6, a full and complete online store), Catwin (v0.7-1, an on-line accounting system) and SuperCart (v3.0, a lightweight shopping cart system). Table 3-1 shows the numbers of tables, PKs and FKs of each database application.

Table 3-1. The experimented PHP database applications

System	# Tables	#PKs	#FKs
OpenDocMan	10	5	7
AFCommerceShoppingCart	22	22	7
Catwin	20	18	8
SuperCart	67	65	15

For evaluation, we manually removed the code related to the KRC constraints enforcement or exception handling from the initial code of these systems. Furthermore, it was found that these systems do not explicitly specify the FKs. DBMS is unable to check referential constraints under this circumstance. In order to completely test our approach, we manually created the referential constraint schemas in accordance with the context of the systems. Basically, if table A includes an attribute that has the same name with a PK in a table B, we added FK definition for that attribute in A.

3.3.3 Results and Discussion

For each system, we used GEHPHP to compute the total number of operations and exception handlings required in accordance with our pre-defined code patterns, and then calculated the percentage of the cases our tool inserted the exception handling code for each PHP system. The statistics are listed in Table 3-2.

To make the data clear, we plotted the above statistics in histograms, as shown in Figure 3-7.

For both ways of handling exceptions, our tool automatically generated and inserted the exception handling code for all the SQL queries that required exception handling. The correctness of the inserted code was manually examined in runtime by triggering the execution of exception handling code. As expected, all the exceptions were handled appropriately by our inserted code, and the error messages were printed correctly, which meant the managed portion of the exceptions was 100%.

Table 3-2. The Exception Handling Requests in PHP database applications detected by GEHPHP

System		OpenDocMan	AFCommerceShoppingCart	Catwin	SuperCart
Insert	#Total	31	21	10	19
	#Exception Handling Required	15	9	7	17
	Percentage*	48.39	42.86	70.00	89.47
Update	#Total	16	65	37	44
	#Exception Handling Required	4	3	3	4
	Percentage*	25.00	4.62	8.11	9.10
Delete	#Total	12	15	10	2
	#Exception Handling Required	2	4	5	1
	Percentage*	16.67	26.67	50.00	50.00
Total	#Total	59	101	57	65
	#Exception Handling Required	21	16	15	22
	Percentage*	35.59	15.84	26.32	33.85

*Percentage = #Exception Handling Required / #Total

Figure 3-7. Histograms for the statistics of the four systems

In this chapter, we propose to automatically insert the exception handling for KRC violations. The main benefit of our approach is that it automatically generates and inserts the code to handle the exceptions of KRC by identifying code patterns without interference of programmers. This lightens the burden of programmers and improves programming efficiency. Secondly, this can avoid human error in coding exception handling and prevent omission or

inconsistent action when handling the same type of exception. This can also enhance the readability and maintainability of the programs and make software evaluation more convenient. Furthermore, our tool can help to complete the code which enhances the quality of the database applications.

However, our approach and prototype tool have the following limitations. It cannot be directly applied to programs written in object-relation model, but fundamentally, our approach is still applicable as the underlying ORM implementations also use SQL queries. Besides this, MySQL is used to set up the database for each PHP system by analyzing the SQL files. Here, SQL file records all the information of database schema. However, currently some systems build tables with CREATE statements hard-coded in PHP source code instead of using SQL files, which is unsuitable for our approach.

3.4 Related Work

Many efforts have been devoted into the area of database research, focusing on the maintaining of the DBMS constraints enforcement efficiently and effectively [12]. A set of quality metrics to measure referential completeness and consistency is proposed by Ordonez and García-García [70]. Madiraju, Sunderraman, Navathe and Wang [62] proposed integrity constraint checking for multiple XML databases. Lee, Pipino, Strong and Wang [50] proposed to use data integrity within the quality management cycle.

However, their works mostly focused on the implementation of constraint violation checking for DBMS. In our previous work [108], for checking the enforcement of integrity constraints, code patterns are used. In this work, we perform KRC at both the DBMS level and the application level to help measure the quality of database applications. Furthermore, notable work was carried out by [8], who proposed to reverse-engineering a vendor's database in order to understand the overall quality of database software so as to aid commercial decision process. Berztiss and Thalheim [6] identified the classifications of the exceptions in information systems and discussed the detection and handling of the different kinds of exceptions. However, few researches have been performed on the exception handling which is automatically inserted and performed to check the violations of integrity constraints of database applications.

Recently, static program analysis has been applied on database applications for several purposes. Pixy is an open-source tool developed by Jovanovic, Kruegel and Kirda to look out for cross-site scripting vulnerabilities in PHP programs [41; 42]. In [93], the authors discussed approaches on generating SQL statement which is securely prepared to prevent SQL injection attacks. Fu and Qian [30] proposed SAFELI, which is a static analysis framework, designed to

identify SQL injection attack vulnerabilities. In this work, bytecode of ASP.NET applications is examined, and for all the SQL query summit points, they used constraint solver to determine whether the SQL query is safe. Huang, Yu, Hang, Tsai, Lee and Kuo [37] proposed to use lattice-based static analysis algorithm to inspect the web applications against security issues for database applications. The code sections identified as vulnerable would be protected by the instrumented code. In [101], the authors proposed an inter-procedural query extraction approach to recover explicit database queries for the program to execute. Dasgupta, Narasayya and Syamala [20] also provided a static analysis approach to extract SQL queries from ADO.NET binaries. These works mainly concentrated on recreation of complete SQL queries which are embedded in database applications.

Our approach goes further than extracting queries. In our proposed approach, constraint violation analysis is used to perform automatic exception handling on the queries extracted from program source code. In our previous work, we extracted the data lifecycle information from the source code of database applications [58]. Furthermore, we proposed to extract attribute dependency graph automatically through analyzing the source code of database applications [55]. The information extracted can be used in the maintenance process especially in the impact analysis upon modification of a database application. The work presented in this thesis focuses on automatic exception handling. Different from our preliminary work [54], this work uses formal transformation rules applied on extended DB-AST. To date, we affirm from our study into related works that no such approach has been proposed.

Program transformation refers to the process that accepts a piece of program and transforms it into a new one [65]. This is usually used in source-to-source compiler which generally makes use of AST as an intermediate representation. For example, in [21], source-to-source compiler is applied to automatically transform C code into parallel programs. Similarly in [53], technique is developed to perform program optimization. To simplify the program transformation process, various domain specific languages are designed, including Stratego [11] and ANLTR [73; 74]. ANLTR also provides grammar rule to facilitate the tree transformation. Unlike other works, such as [63] which targeted at other applications, our approach focused on automatic SQL query pattern identification and exception handling code insertion by using expanded AST, which can be viewed as a program transformation process. Although we verified our approach with experiments on applications written in PHP, our approach can be applied on applications in other languages, including object-oriented, multi-layered systems implemented in Java.

3.5 Conclusion

In this chapter, we depicted the problem of KRC enforcement in database applications and the problem of exception handling of KRC violations. We proposed an approach to include SQL query semantics in the classic AST. Based on the defined KRC violation patterns, our approach can automatically analyze the SQL queries and then determine the KRC violation patterns. Each code pattern that requires exception handling together with the code to be inserted as precondition or post-condition is represented as a transformation rule. We provided two alternative ways to handle the exception, one to handle it solely by program without DBMS enforcement and the other to handle it when it is detected by DBMS. Moreover, we implemented a tool and conducted four case studies to evaluate the proposed approach. The conclusions drawn from the studies were also presented.

Chapter 4

AIDING MAINTENANCE OF DATABASE APPLICATIONS THROUGH EXTRACTING ATTRIBUTE DEPENDENCY GRAPH²

Database is a major component of many software systems and database applications are widely applied in various areas. Due to the need for more functionality, integrity and other operating characteristics, database applications are becoming more and more complicated, and this rising complexity calls for more frequent updating in the application.

In practice, the database schema is continuously changing because system requirements keep changing in time. Moreover, since the data that is kept within the database is generally dynamic, the programs must always be modified to maintain the changeable data. An inadequate change on schema may possibly break the original system [13]. However, due to necessities and benefits, changes on schema are widely applied on database applications [3]. Over the modification process, impact analysis of changes must be conducted to identify their potential consequences. However, manual assessing of these effects is a tiring and complicated process [48], and can be frequently incorrect. In addition, there are many dependencies between attributes maintained in a database which can introduce many obstacles for maintaining database applications. Considering these issues, it is essential to provide simple to use and straightforward information to aid the maintenance of database applications, especially by performing impact analysis after modification. However, little work [31] has been done in the field of software maintenance research targeting specifically on database applications. Hence, in this chapter, we address this issue.

In general, graphs provide an intuitive and clear way to depict the relationships between entities and have been utilized in software engineering for a long time. A variety of graphs, such as CFG, function call graph, program dependence graph, have been introduced to aid software maintenance processes, and their usefulness has also been demonstrated. However, to

² This chapter is the author's version of the work. It is posted here by permission of Journal of Database Management (JDM). Not for redistribution. The definitive version was published in Liu, Kaiping, Hee Beng Kuan Tan, and Xu Chen. "Aiding Maintenance of Database Applications Through Extracting Attribute Dependency Graph." *Journal of Database Management (JDM)* 24.1 (2013): 20-35.

the best of our knowledge, little research has been delved in the use of graphs to assist the maintenance of database applications.

In this chapter, we propose a novel graph called attribute dependency graph to demonstrate the dependencies between attributes in a database application and also the programs involved. We propose an approach to extract dependencies between attributes and then construct attribute dependency graph by analyzing the source code of database applications. The extracted graph can be used to aid maintenance processes particularly in the impact analysis upon modification of a database application.

Contributions and Results

- We propose attribute dependency graph to reveal the dependencies between attributes in a database application and also the programs involved.
- We propose an approach to automatically extract the attribute dependency graph out of a database application from its source code through inter-procedural static program analysis.
- We build a tool to implement the proposed approach for PHP-based database applications.
- Case studies have been conducted to examine the impact of changes made to these database applications, including dropping of attributes, change to attributes and changes to programs.

This chapter is organized as follows. Chapter 4.1 introduces the attribute dependency graph and how we extract the attribute dependency graph. Chapter 4.2 presents the prototype tool. Chapter 4.3 reports our evaluation through case studies. Chapter 4.4 presents the related work. Chapter 4.5 concludes the chapter.

4.1 Extraction of Database Attribute Dependency Graph

Within a database application, a set of database attributes could be referenced by an SQL query in a program to decide whether other database attributes could be inserted, updated or deleted. Moreover, values of some attributes could also be used in the SQL query to assign other attribute values. When either of these two cases occurs, we note that dependencies exist among attributes. For every change that has been made to a database attribute, we must identify and investigate the change's impact to other attributes that depend on the changed attribute and respond accordingly with necessary actions.

To support the above-mentioned process, we introduce attribute dependency graph to represent the dependencies between attributes. Our attribute dependency graph displays some

similarities to the data dependency graph (DDG) for program analysis in that both of them illustrate the dependencies between data. However, the DDG only indicates the data dependency for the variables defined in a program but not the database attributes.

4.1.1 Defining Attribute Dependency Graph

In this chapter, a program is defined as a basic independent executable unit in a database application. In the attribute dependency graph, a program is represented with a rectangular node, while an elliptical node represents a database attribute that is referenced, inserted, updated or deleted by a program. An edge from an attribute to a program indicates the program references to the attribute, while an edge in the opposite direction indicates that the program modifies the attribute. Here, the modification could be an insertion, update or deletion. Whereas incoming edge to a program is labeled with a reference number to identify a reference to an attribute made in the program, each outgoing edge from a program is labeled with both the type of modification and the reference numbers indicating the references identified by the incoming edges to the program. Figure 4-1 shows the basic notations of attribute dependency graph.

Figure 4-1. Basic notations of attribute dependency graph

Since PHP is a widely used scripting language for developing web application, many database applications are developed using PHP. This thesis adopts systems written in PHP for experiments and demonstrations. However, the proposed approach is general enough to be applied to any database programs. As a demonstration, consider a simple scenario of a bank management system. Suppose one user desires to withdraw \$100 from his bank account while another wants to cancel his bank account. In order to implement both transactions, there should be one Update and one Delete SQL operation. Assume that these two operations are executed by programs P and Q respectively, and there are three attributes “AccountNo”, “Name”, and “Balance” in the table “Customer”. The following PHP code implements this scenario.

```

// program P:
<?PHP
$withdrawal = $_REQUEST['withdrawal_amount'];
$accountNo = $_REQUEST['accountNo'];
$result = mysql_query("SELECT Balance FROM Customer WHERE AccountNo=" .
    $accountNo);
$row=mysql_fetch_assoc($result);
$balance=$row["Balance"];
if( $balance > $withdrawal )
    mysql_query("UPDATE Customer SET Balance=Balance-". $withdrawal . " WHERE
        AccountNo=" . $accountNo);
else
    die("Not enough balance");
?>
.....
// program Q:
<?PHP
$name=$_REQUEST['accountNo'];
mysql_query("DELETE FROM Customer WHERE AccountNo=" . $accountNo);
?>

```

In program P, after the user enters the amount to be withdrawn and his account numbers, his account balance is first checked to ensure that it is sufficient for this withdrawal. If yes, the new balance is updated. Otherwise the program is terminated by calling the “die” built-in function. In the first query, the value of the attribute “Customer.Balance” will be decided by itself, and the target record to be updated will depend on the attribute “Customer.AccountNo” in the “WHERE” condition. Hence, there are two dependencies in this SQL query. In the second query, the record to be deleted from “Customer” is decided by the attribute “Customer.No”, so three more dependencies are involved. The corresponding attribute dependency graph of the above-mentioned scenario is shown in Figure 4-2.

Figure 4-2. Attribute dependency graph

4.1.2 Control and Data Dependency Analysis

In database applications, data is generally dynamic and manipulated by SQL queries. Before extracting the attribute dependency, all SQL queries must first be located and the following information identified: 1) attribute names and table names which are included in the

SQL queries and the implied dependence relationship; 2) the operation type of the SQL queries. SQL queries in the source code are commonly formed as string variables, which could be then concatenated by multiple variables. The formed queries are then passed to the specific query functions. Since the values of string variables involved in the SQL query commonly vary according to different conditions, the actual value of the variable is subject to different program paths chosen at runtime. Figure 4-3 shows an example.

```
1. <?php
2. if (...)
3.     $attr_to_update = 'col1';
4. else
5.     $attr_to_update = 'col2';
6. $sum='(SELECT sum(col) FROM tb2)';
7. mysql_query('UPDATE tb1 SET ' .
    $attr_to_update.'='.$sum.' WHERE col3=...');
8. ?>
```

Figure 4-3. PHP code snippets - test.php

During the extraction process, in order to resolve the variables' values used in the SQL queries, the CFG is first computed, and based on that, inter-procedural data dependency analysis is performed. The standard CFG for each program is computed and the nodes that contain SQL query function calls are identified. Figure 4-4 shows the CFG for the code of test.php in Figure 4-3.

For each node containing one or more SQL query function calls, inter-procedural analysis is performed to extract all the execution paths starting from the entry node. For example, from the CFG in Figure 4-4, two paths can be extracted for node 6. One is "Entry → 2 → 3 → 5 → 6 → Exit" and the other one is "Entry → 2 → 4 → 5 → 6 → Exit". Data dependency analysis is performed specifically for the variables involving in the functions' parameter for each extracted path. A backward trace is made along the path for each variable to record nodes that the variable is data dependent on until the value of the variable can be determined. When this is achieved, the value of the variable is propagated downward along the path to the query function. All the values are then concatenated in the same order as they appear in the parameter of the function call and the SQL queries are constructed.

Figure 4-4. CFG for the code of test.php

For the example in Figure 4-4, from the first path, it could be found that the value of `$attr_to_update` is "col1", and the value of `"$sum"` is "(SELECT sum(col) FROM tb2)". Consequently, one SQL query is generated: "UPDATE tb1 SET col1 = (SELECT sum(col) FROM tb2) WHERE col3=...". Similarly, for the second path, another SQL query is produced: "UPDATE tb1 SET col2 = (SELECT sum(col) FROM tb2) WHERE col3=...". These queries as well as the associated file name of the programs are recorded for further analysis.

4.1.3 Attribute Dependency Analysis

After the possible values of the variables used in the query function are retrieved, the set of SQL query strings are formed. Each SQL query is then parsed according to the standard SQL grammar specification. The operation type of the query and the attribute names are identified. Next, the dependence relationships are identified by dividing the query into the depending and depended parts. For example, the three parts of the first query "test.php" are "UPDATE tb1 SET col2", " (SELECT sum(col) FROM tb2)" and "WHERE col3=...", and the attributes in the first part are data dependent on the attributes in the second part, and control dependent on the attributes of the third part. Thus, it is identified that `tb1.col2` is dependent on `tb2.col1` and `tb1.col3`. The dependency is then recorded in an ordered tuple (`{attribute set1}`, `{attribute set2}`, `<program, operation type>`), where the attributes in the first set are dependent on the attributes in the second set, and the third element of the tuple is an ordered pair recording the program containing the SQL query and the operation type.

Attributes that are explicitly given can be recorded directly. For the function call in line 7 in `test.php`, two ordered tuples are finally extracted: (`{tb1.col1}`, `{tb1.col3, tb2.col1}`, `<test.php, UPDATE>`) and (`{tb1.col2}`, `{tb1.col3, tb2.col1}`, `<test.php, UPDATE>`). For those attributes that are given implicitly in the queries, such as the `*` symbol in SQL operations that omit the

attribute list, the meta data of database schema is first retrieved from the database management system and a map is built between table names and their attributes involved in the tables according to the schema's definition. Whenever an implicit attribute list is encountered, the table name is used to retrieve attribute names from the map. This process can ensure the precision of the analysis.

4.1.4 Construction of the Graph

After all ordered tuples are collected, the attribute dependency graph is generated accordingly. For each dependence tuple, a check is first performed to detect whether there are already nodes for every attribute in the two sets. For those attributes that have not been drawn on the graph, elliptical nodes are drawn to represent them. This process is applied to each program in the database application, and the programs are represented with rectangular shapes. After that, since all attributes in the first set of the tuple are dependent on those in the second set, directed edges with labels as described in Chapter 4.1.1 are drawn to connect from the nodes representing the second set of attributes to the program node. Each pair of dependence relationship is assigned with a unique serial number labeled on the edge. Then directed edges with corresponding labels are drawn from the program node to the first set of attributes nodes. Figure 4-5 shows the attribute dependency graph for the PHP code snippets in Figure 4-3. There are totally four dependence pairs in this graph.

Figure 4-5. Attribute dependency graph for program in Figure 4-3

4.2 Prototype Tool

Since a significant number of database applications are implemented in PHP, we implemented our prototype tool in Java which targets specifically PHP-based database applications. We used Pixy [40], an open-source PHP analyzer to parse the PHP source code and DOT (<http://www.graphviz.org/doc/info/lang.html>), a graph description language to draw and display the graphs.

The tool operates as follows. First, database schemas that have been defined in the database applications are identified by using the metadata accessing methods, such as the method `getMetaData` provided by JDBC. Then, the metadata, the map between table names and attributes, is built. Subsequently, each PHP program is parsed and the corresponding CFG is computed. The tool then performs control and data dependency analysis on these CFGs. Based on this, the SQL queries contained in the programs are extracted. Attributes and their dependence relationships are discovered through queries analysis, and the tuples containing the dependency information as described in Chapter 4.1.3 are formed. Using these tuples, the attribute dependence graph for the whole system is drawn by outputting the DOT code. Finally, the graph is displayed by invoking external graph displaying tools, such as `dotty` in the Graphviz package; or the DOT code is interpreted to generate general image format like PNG. Figure 4-6 shows the major steps of the prototype tool.

Figure 4-6. The workflow of the prototype tool

4.3 Case Studies

Once the database schema is defined, it would be difficult to ensure the system's correctness and efficiency because schema changes may generate coupling issues between database schema and the source code [3]. When the database schema is changed, the implementation process requires thorough examination and identification of all the potential impact.

4.3.1 Testing Dataset

Three PHP systems have been featured in our case studies: ECShop between version 2.1.2 and 2.1.5, Frontaccounting between version 2.1 and 2.3 and the most recent two releases of a student league membership management system. These open-source database applications cover several domains. ECShop is a web-based online shopping system written in PHP which

uses a relational database (currently MySQL) to store data. Frontaccounting is a professional web based powerful system for the entire ERP chain. The membership system has been in operation for many years and is used to schedule meetings, events and so on. The prototype tool has been executed on these systems to construct attribute dependency graphs for each of the systems. These case studies examined the impact of changes made to these database systems, including dropping of attributes, change to attributes and changes to programs. Table 4-1 displays the statistics about the number of attributes are dropped and changed, and also the number of affected attributes dependencies and programs associated with these attributes respectively.

Table 4-1. Overview statistics of three systems

Systems	ECShop	Front-accounting	Membership Management
#Dropped Attributes	9	16	2
#Dependencies Associated with Attributes Dropped	24	29	9
#Programs Associated with Attributes Dropped	12	15	4
#Changed Attributes	36	37	6
#Dependencies Associated with Attributes Changed	82	85	19
#Programs Associated with Attributes Changed	17	33	14

From Table 4-1, it can be observed that database schemas are changed frequently and there are many attribute dependencies and correspondingly-affected programs. It is therefore a laborious process to perform manual analysis, and our proposed attribute dependency graph would benefit the process significantly. We will show the evaluation case studies related to the impact of changes made to these database applications, including dropping of attributes, change to attributes and changes to programs.

4.3.2 Dropping Attribute

During database application development and maintenance, developers may occasionally realize that the previous schema is not well organized and some attributes are useless and redundant for the system, or that some attributes have become obsolete because the schema does not meet the new requirement. Under these situations, the schema should be modified to drop these redundant attributes. However, for each attribute that is being dropped, there will be direct impact on programs which reference the attribute. All SQL queries which include this removed attribute would become erroneous since it no longer exists. In addition to this, there could also be potential risks for those attributes that are dependent on the dropped one. The proposed attribute dependency graph can be used to aid the impact analysis for this case.

A case study has been conducted to demonstrate the graph’s usefulness by analyzing an open-source system ECShop. ECShop is a web-based online shopping system written in PHP which uses a relational database (currently MySQL) to store data. This system has been under active development for several years and we noted that frequent updates have been made on the database schemas. In the upgrade package of version 2.1.5, there are relatively large amount of database schema modifications, including 9 dropped attributes in 5 different tables. Table 4-2 shows the list of attributes that are dropped.

Table 4-2. Dropped attributes of ECShop

Dropped Attributes	Table	Attribute
	ecs_bonus_type	send_count
	feedback	Reply
	ecs_sessions	Expireref
	ecs_goods	can_handsel
	ecs_goods	fitting_price
	ecs_goods	is_linked
	ecs_goods	is_basic
	ecs_cart	is_gift
	ecs_category	is_leaf

Figure 4-7. Attribute dependency graph of ECShop

An inspection was made on the source code of version 2.1.2 using our tool, and the corresponding attribute dependency graph for the system was built. Due to space constraints, Figure 4-7 only shows the generated graph related to the attribute “is_gift”. From Figure 4-7, it can be observed that all attributes of table “ecs_cart” are dependent on the dropped attribute “is_gift” with the DELETE operation. Since “is_gift” is dropped, this DELETE operation is no longer valid, and thus should be modified or removed. But developers must also conduct

verification of this. For example, if the requirement states that the attribute should have the function of deletion, simply removing the query would suggest that there is no way to delete records with specified values of attribute “is_gift” in this system, so developers must provide DELETE operation for them in other programs. Caution must be exercised when checking for attributes that are dependent on “is_gift” to ensure that they conform to the system requirement or specification. Similarly, a check should be conducted for dependencies affected by other attributes listed for dropping in Table 4-2 against the system requirement and specification.

Through our attribute dependency graph, developer and maintainer can achieve an overall picture of the impacts caused by the dropping of some attributes, hence allowing them to assess feasibility as well as estimate the amount of effort required for the change without necessitating manual review of the source code. This is crucial if detailed documentation of the application is absent.

4.3.3 Changing Attribute Definition

Changing of attribute’s definition, such as data type and constraint, is another common database modification behavior. Over the process of development and maintenance, the original data type of some attributes may become obsolete for the new system requirements. The definition must therefore be amended in order to handle new types of data. For example, in a database application, there is an attribute originally defined as UNSIGNED SMALLINT. However, as data size expands, the developer or maintainer realizes that the number of items is likely to exceed the maximum of UNSIGNED SMALLINT. It becomes necessary to change the data volume of attribute to larger integer type, such as UNSIGNED INT. Another possible definition change is the modification of string field length, e.g. from CHAR(20) to CHAR(15). In these cases, there could be potential hazardous results for loss of data, thus source code reexamination is needed to ensure the SQL queries which involve the changed attributes produce consistent results as required. Our proposed graph is also able to aid this process.

An analysis was made on the open-source system Frontaccounting to demonstrate this process. Frontaccounting is a PHP-based accounting system covering the entire ERP chain. Through inspection of database upgrade from its version 2.1 to version 2.3, we found that there are totally 37 definition changes on attributes from 17 different tables. Table 4-3 shows a part of the attributes whose definitions have been changed.

In order to safely perform these changes, maintainers need to check all SQL queries in which these attributes are involved and determine whether there would be potential errors caused by changes. Thus we applied our prototype tool on the CVS snapshot of version 2.1,

and built the attribute dependency graph of the system. Due to space limitation, Figure 4-8 only shows part of the whole graph closely related to the attribute “order_no”.

Figure 4-8. Attribute dependency graph for FrontAccounting

Table 4-3. A part of changed definitions of attributes of Frontaccounting

FrontAccounting			
Table	Attribute	Definition before v2.3	Definition in v2.3
sales_order	order_no	INT(11) NOT NULL AUTO_INCREMENT	INT(11) NOT NULL
buget_trans	account	VARCHAR(11)	VARCHAR(15)
chart_master	account_code	VARCHAR(11)	VARCHAR(15)
chart_master	account_code2	VARCHAR(11)	VARCHAR(15)
tag_associations	record_id	VARCHAR(11)	VARCHAR(15)
chart_class	cid	INT(11)	VARCHAR(3)
chart_types	id	INT(11) NOT NULL AUTO_INCREMENT	VARCHAR(10)
chart_types	class_id	TINYINT(1)	VARCHAR(3)
.....			

From Figure 4-8, it is easy to locate the programs in which the changed attributes are dependent on by other attributes. For example, for the attribute “order_no” in table “sales_order”, there is a DELETE operation takes place in program “fiscalyears.php”, in which all attributes in table “sales_order” depend on the attribute “order_no”. Maintainers thus need to check whether the change leads to different behavior of this deletion and ensure that when this query is executed, the right records are deleted. In the same way, maintainers can check the dependencies in “sales_order.view.php” as well as “sales_order_db.inc” relate to attribute “order_no”.

Similar process is applied on attribute “cid” in table “chart_class” and “id”, “class_id” in table “chart_types”. The data type of attribute “cid” is changed from INT(11) to VARCHAR(3). Also the data type of “class_id” in “chart_types” is changed from

TINYINT(1) to VARCHAR(3). It was discovered from the generated attribute dependency graph that in the programs “gl_db_account.php” and “gl_db_account_types.inc”, there are several attributes depending on these three changed attributes. Examination should be undertaken to ensure that the following errors would not occur: 1) if the values of the depending attributes are assigned by the values of the changed attributes, their data types are not compatible; 2) if the changed attributes are used in condition tests, the conditions are not consistent to the specification.

With our attribute dependency graph, the developer and maintainer can determine the adequacy of the ongoing change by verifying whether the depending attributes associated with these changed attributes still get provided with valid handling.

4.3.4 Changing Program

During the maintenance process, changes to database operations that are initiated from programs are very common. Our attribute dependency graph can help in determining the impact of such changes with respect to databases which are the major components in such systems.

The last case study was conducted on a PHP-based membership management system for a student league. Recently, the members who use this system have experienced some inconvenience in the function for updating the weekly most suitable timeslot for meeting. The reason is that the input screen to update the frequently changed timeslot is together with updating of address, phone number etc., which actually seldom change. Hence, each time when a member updates the weekly suitable meeting timeslot, those fields that seldom change are also displayed and required to be filled. This is surely troublesome and now the application requests the removal of all other fields from this screen, in order that this screen will be solely for the updating of suitable weekly meeting time slot.

The original program segment containing the UPDATE query is as follows:

```
.....  
mysql_query("UPDATE meeting SET starttime=..., endtime=..., address=..., phone_number=... WHERE  
group_id=...");  
.....
```

The corresponding attribute dependency graph related to these attributes is shown in Figure 4-9.

Figure 4-9. Attribute dependency graph for the student league membership management system

After analyzing the attribute dependency graph, it could be surmised that these attributes could only be updated within the program “update_meeting_time.php”. Consequently, when all attributes other than the “starttime” and “endtime” have been removed from this screen, there is no other program updating the “address” and “phone_number” attributes. After further investigation into the system, it was discovered that although these attributes seldom change, they still need to be kept dynamic and changable. Therefore, in addition to removing these attributes from the screen, a new screen also must be provided for updating these attributes.

With the assistance of our graph, the impact of those changes can be identified and assessed in this way. During the development and maintenance of database application, changes on programs may frequently affect some database operations. Before making a decision to modify or remove the SQL operations in certain programs, developers or maintainers can first refer to the attribute dependency graph of the system and check whether the affected attributes would lead to inconsistency or incompleteness of functionality.

4.3.5 Discussions

In this chapter, we propose to generate attribute dependency graph from source code of database applications. We believe that this approach is general enough to be applicable on any programming languages. Furthermore, this approach can be applied in any database application including cases that have recursion and aliasing, all of which would be further examined and evaluated in later implementations of our system.

From the case studies, we can see that the proposed attribute dependency graph can assist the maintenance process by providing accurate visual description of attribute dependencies. Moreover, as impact analysis of modification in database applications is time consuming, the attribute dependency graph can lighten the burden of the programmer and improve the efficiency of maintenance. In addition, the approach can avoid human error in examining and identifying of all the potential impact. Last but not least, the attribute dependency graph can be also used to assess feasibility as well as estimate the amount of effort required for the change without necessitating manual review of the source code.

However, there are a few limitations. First of all, the approach is used to analyze the SQL queries in the source code and not meant for dealing with queries outside source code such as stored procedure and dynamic queries from user input. Secondly, the approach has yet to verify its effectiveness on large quantity of source code. It must be acknowledged that, when the scale of source code goes up, representation of the graph may become more difficult and higher levels of abstractions are needed.

4.4 Related Work

Impact analysis is an important issue in the field of software engineering, and much research has been delved into exploring software change impact analysis [9; 48]. Earlier works have been focused mainly on object-oriented database, but little work has been carried out on enhancing the impact analysis of database schema. Maule, Emmerich, and Rosenblum [64] have proposed an approach for impact analysis of database schema changes. But their approach is different compared to the approaches proposed in this thesis in that it considers mainly the database queries in the programs, but not the dependencies involved in the queries.

Recently, a number of studies have gone into the investigation of extracting database queries from database applications by conducting program analysis. Gardikiotis and Malevris [32] proposed to aid database impact analysis and regression testing using program slicing and program dependency graph, though this method primarily analyzes the program flow dependencies. Gould, Su, and Devanbu [34] came up with an approach for predicting the content of strings which are parameters for the Java JDBC library methods by using string analysis. Besides the above works, Tan, Zhao and Zhang [92] made use of multiple linear regression model to estimate the LOC for information systems from their conceptual data models. Our approach differs from these previous works by using data flow analysis to recover the queries in the program, which can be used for constructing attribute dependency graph. The graph can then be used to aid impact analysis on several aspects.

Several previous works have been conducted on restoring and expressing the inner mechanisms of database applications. Chan, Siau and Wei [14] studied the effect of entity-relationship versus relational models, and textual versus visual query languages for user-database interfaces. Their method can assist database developer to choose and design effective and efficient interfaces for end users. Tan, Ling, and Goh [90] proposed a new approach to extract common data dependencies with program analysis. The designed data dependencies can therefore be used for maintaining and reengineering of database applications. In 2004, another approach was put forth by Tan and Thein [91] for recovery of provisions and designs of transactions to automatically rectify post-transaction userinput error (PTUIE) in database applications. In our previous work, we extracted the data lifecycle information from the source code of database applications based on the concept of data lifecycle [58]. Moreover, Create, Retrieval, Update and Delete (CRUD) tables have been employed to provide summaries of the tables and attributes' usage, but these approaches do little to disclose the attributes dependency relationships [96; 97]. Nagy, Pántos, Gergely, and Beszédes [67] constructed a CRUD graph which indicated the dependencies between procedures or database tables. But their graph stopped short of addressing the dependencies among attributes and was not applicable in impact analysis or other maintenance process. Besides the method proposed by Gardikiotis and Malevris [32], program slicing has also been engaged to build a system dependency graph that can recover dependencies among embedded database instructions in the programs [18; 36]. Unlike the approach proposed in this thesis, none of these methods, however, has considered the attribute dependencies in the database and their usage for impact analysis. Our approach explores the relationships among database attributes in attempt to reveal the underlying business specification.

4.5 Conclusion

The database application is always changing for different purposes, and for every modification done to the application, we must take into account what effects the change will have on the overall system. This process is too difficult and error-prone to be performed manually. Additionally, the sheer number of attribute dependencies in the database contributes to difficulties in the maintenance process of database applications. It is for this that we propose the attribute dependency graph to reduce problems in the maintenance process.

We propose the attribute dependency graph which illustrates the dependencies between attributes in a database application and the programs involved. We also propose an approach to extract and construct the attribute dependency graph through the analysis of the source code of database applications. Furthermore, we have developed a tool to implement the proposed

approach for PHP-based database applications. Case studies have also been conducted to demonstrate the use of our proposed approach.

The main application for the proposed approach outlined in this chapter is to aid in impact analysis when the database is changed. The attribute dependency graph can identify all attributes and programs affected by changes made to the database. On one hand, it can lighten the burden of the maintainer and improve the efficiency of maintenance activities. On the other hand, it can also avoid human errors caused by manually assessing the effects.

Chapter 5

DETECTING ANOMALY IN THE USAGE OF DATABASE ATTRIBUTE³

In database applications, adequate operations should be provided to maintain the persistency of the data. Any missing, redundant or inconsistent operations performed on database attributes would indicate anomaly, which implies error and incompleteness of a database application. Any instance that cannot be classified as normal behavior is known as an anomaly.

The process of searching for patterns that deviate from common behavior is known as anomaly detection. The goal of anomaly detection is to develop a system that would automatically detect such patterns. Once an anomaly is detected, developers can perform investigation to take corrective actions. The PHP code snippet shown in Figure 5-1 from an industry system is an example of anomaly in database operations:

```
<?php
function do_query($query)
{
    return mysql_query($query);
}

$sql = "SELECT family_name as fn FROM tb_user_account WHERE user_id=";
if (isset($_POST['USER_ID']))
{
    $sql .= $_GET['USER_ID'];
}
else
{
    $sql .= '0';
}
$result = do_query($sql);
$data = mysql_fetch_array($result);
$sql = "UPDATE tb_user_info SET family_name=" . $data['fn'];
do_query($sql);
?>
```

Figure 5-1. PHP code example

In the example in Figure 5-1, the attribute “family_name” is selected from the table “tb_user_account” in order to update the attribute in table tb_user_info. However, from the

³ This chapter is the author's version of the work. It is posted here by permission of Knowledge Systems Institute. Not for redistribution. The definitive version was published in K.P. Liu and H.B.K. Tan, “Detecting anomaly in the usage of database attribute” in *Proceedings of 26th International Conference on Software Engineering & Knowledge Engineering*, pp. 705-710, July 2014.

database schema, we found that the second table requires non-null value for the attribute “family_name” but the first table does not have such requirement. Furthermore, from investigation of the code of the whole system, there is no INSERT operation to define the value of the attribute “family_name”. So the result of the SELECT operation in the above example contains a null value of the attribute “family_name”. However, before the attribute “family_name” is used for updating, the value of it should be defined. Hence, the execution of the UPDATE operation would cause exception and even runtime failure. It is desirable to be able to detect such kind of anomaly use of data attributes.

Traditionally, the classical classification algorithms used in anomaly detection require labeling instances in a data-set to train classification models. However, effective classification typically requires a large amount of training data, and manual creation of these training data proves to be tedious. Furthermore, normal behavior changes from time to time and what is known as normal behavior at one moment may not be so later on. For these reasons, classification-based anomaly detection algorithms which rely on labeled data are often inaccurate and highly expensive. These difficulties are compounded by the fact that labeled data is not always available.

Within the study of data mining, clustering technique is among the most practical unsupervised learning techniques. It assigns a set of objects into clusters by maximizing intra-class similarity and minimizing inter-class similarity. Clustering can be applied to identify interesting distributions and to discover useful patterns in the underlying data. Anomaly detection is best applied using clustering because these techniques can be applied to unlabeled subjects. Figure 5-2 demonstrates how anomalies can be identified using clustering. This dataset consists of two clusters which are both normal, N1 and N2, to which most observations belong. O1, O2 and O3 are thus considered as anomalies because they are too far away from both of the two normal clusters.

This chapter proposes an approach to detect anomalies in the use of database attributes by means of abstracting and characterizing database operations performed in database transactions. We present an unsupervised, clustering-based anomaly detection algorithm, which takes as inputs a set of unlabeled database attributes and finds anomalies within them.

Figure 5-2. An example of outliers in a data set with 2 dimensions

Two general assumptions about the data have been made in our unsupervised anomaly detection algorithm. The first one assumes that the number of normal attributes is much greater than that of the anomalous attributes, meaning that larger clusters should be formed by normal attributes than the anomalous attributes. The second assumption is that the anomalous attributes and the normal attributes are qualitatively different, thus implying that it is impossible for them to be grouped into the same clusters. Any anomalous attributes in the dataset would therefore appear to be outliers due to their rarity and abnormality.

Our approach groups the database attributes together into clusters using a distance-based metric. Once the database attributes are clustered, we identify small clusters and label them as anomalous clusters. We perform cross-project validations on open-source database applications to verify our approach. The results show that our approach is able to detect many types of anomalies with an average detection rate 85.5, while maintaining a low false positive rate.

Contributions and Results

- We characterize operations performed in database transactions on database attributes; we extract a feature vector from code for each attribute.
- We propose a clustering-based approach which analyzes the feature vectors to automatically detect anomalies in the usage of database attributes.
- The evaluations on both industrial and open-source database applications show that our approach is able to detect many types of anomalies in the usage of database attributes with a high detection rate (92.8% on average), and a low false positive rate (0.57% on average).

The chapter is organized as follows. Chapter 5.1 introduces how we characterize database operations and the extraction of the feature vector for attribute usage. Chapter 5.2 describes

our proposed approach. Chapter 5.3 evaluates the proposed approach and reports the experiment results. Chapter 5.4 discusses the performance of our approach and its variations. Chapter 5.5 presents the related work. Chapter 5.6 concludes the chapter.

5.1 Characterizing Database Operations Using Feature Vectors

Since transaction is the atomic processing unit in a database application, we characterize the database operations performed on database attributes on a per-transaction basis.

5.1.1 Database Operations

We classify the database operations into the following types:

- Create (C): A value of an attribute is inserted.
- Null Create (NC): A record that contains the attribute is inserted without defining the value of the attribute.
- Control Update (COU): The value of an attribute is updated by a new value that is not influenced by the existing attribute value and inputs from user and database.
- Overriding Update (OVU): The value of an attribute is updated to a new value that is influenced by user input but is not influenced by the existing value of the attribute.
- Cumulating Update (CMU): The value of an attribute is updated to a new value that is influenced by the existing attribute value.
- Delete (D): The value of an attribute is deleted as a result of deletion of a record containing it.
- Use (U): The value of an attribute is used to support the insertion, updating or deletion of other database attributes or output to the external environment. It is not considered under this case if the value of an attribute is used to update itself.
- Other Update (OU): If there are multiple database operations performed on a database attribute in one transaction, the database operation is classified as Other Update.

We characterize the usage of a database attribute using an eight-element Boolean vector $[m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8]$, where $m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8$ denote the existence of a transaction in which an operation performed on the attribute is of type C, NC, COU, OVU, CMU, D, U or OU respectively. This vector is called feature vector for attribute usage. For example, if static analysis finds the following three paths in the CFG through a database application that performs database operations of type C, OVU and D on attribute A

respectively, the feature vector for attribute A is [1,0,0,1,0,1,0,0]. Based on the feature vector, we perform data mining through clustering to discover the anomalies in the use of attributes of database applications.

To automatically extract the characteristics of database operations performed on each attribute in a transaction, we perform static program analysis on each program in a database application. We analyze the database operations performed in each path of a database program. During the analysis of paths, we traverse the loop body only once. As such, the number of paths will be finite in our analysis.

5.1.2 Extraction of Feature Vectors for Attribute Usage

To automatically extract the database operations performed on each attribute in a transaction, static program analysis is performed on each path in a database application. Since PHP is a widely used scripting language, this chapter uses PHP programs as subjects for demonstrations and experiments. However, the approach can be applied to any database application as it is based on basic control and data dependency analysis with regard to standard SQL statements.

In order to obtain the feature vectors for database attributes usage, we need to analyze the SQL queries in a database application. The SQL queries in the source code are commonly formed by dynamically concatenating several string literals and variables. As can be seen from the code in Figure 5-1, the actual values of the first query could be different according to the if-condition which cannot be decided during static analysis. In addition, the query is passed as parameter to another function `do_query` for execution. Note that in PHP, those lines of code that are not within a function scope can be viewed as in one virtual MAIN function.

Figure 5-3 shows the inter-procedural CFG for the code snippet in Figure 5-1. In Figure 5-3, the box on top represents the control flow of the virtual MAIN function while the bottom box represents that of the “`do_query`” function. The solid and dashed lines represent intra-procedural and inter-procedural control flows respectively. There are two paths that perform database transactions, both of which start from the MAIN’s entry point, and reach the SQL execution function “`do_query`”. However, one of the paths goes through the first branch of the if-condition node while the other path goes through the other branch. Hence, the actual queries executed in these two transactions are different.

We analyze the CFG on the per-path basis. However, enumeration of all paths on the inter-procedural CFG is not computationally affordable. Therefore we follow the technique used in structural program testing by generating a set of basis paths using the baseline method proposed in [100]. The basis paths cover all branching nodes as well as the corresponding

branches. Then for each path in the basis set, whenever we encounter a query execution function like “mysql_query”, the definition of every part of its parameter which is a query string is retrieved. For each part of the string concatenation, the literals are replaced by their actual values, and those variables whose values are not statically known are replaced by placeholders in order to preserve the SQL grammar. Finally, all the parts of query strings with replaced values are connected in their original order to form the extracted queries. One transaction typically contains a set of SQL queries.

Figure 5-3. Inter-procedural CFG for PHP code snippet in Figure 5-1

As an example, for the code in Figure 5-1, the extracted queries of the SELECT operation would be: 1) "SELECT family_name as fn FROM tb_user_account WHERE user_id='__'"; 2) "SELECT family_name as fn FROM tb_user_account WHERE user_id=0".

After the queries are extracted, we analyse each query to obtain the characteristics of database operations using an SQL grammar parser. All the CREATE TABLE queries are first parsed. Then, we analyse the queries according to their operation types as follows:

- **SELECT:** The SELECT query is parsed, table aliases are identified and restored by the actual table names, and the attributes are identified. The attribute names are extracted from the select list, JOIN expressions as well as the WHERE clause. The star-shorthand “*” is regarded as referencing all attributes.
- **INSERT:** After parsing, the table name is identified. We then check whether there is column list in this query. When no column list is provided, it is assumed that values of all the attributes in this table are to be inserted. Those attributes declared in the schema as “auto incremental” or having not-null default values are also viewed as inserted by the query.
- **UPDATE:** We not only collect the attribute names that are updated in one query, but also identify the update pattern for each attribute. We analyze the value string to determine the update type, i.e. either COU, OVU or CMU for the attribute. For example, the assignment expression `name=$_POST['username']` is an example of OVU, whereas the expression `balance=balance-$withdraw` is an example of CMU.
- **DELETE:** We identify the table name, and mark all the attributes of this table as “Delete”.

Besides, the attributes in the WHERE clause are characterized as “Use”. If an attribute has multiple operations in this transaction it will be marked as OU in this transaction; otherwise, it will be marked as one of the other seven features as described in Chapter 5.1.1. After all queries in one transaction are parsed and analyzed, we check each attribute that is involved in this transaction to determine whether it has unique operation or not.

After all transactions are analyzed, the characteristics of database operations performed on each attribute are merged together to generate the feature vector for database maintenance and usage. For example, if an attribute experiences the type “Create” for at least one time, the first element of the feature vector would be 1; otherwise, it is 0. If there is no transaction that performs database operations on an attribute, the feature vector of the attribute is set to [0, 0, 0, 0, 0, 0, 0, 0]. In this way, we extract feature vectors for all the attributes in a database application.

The algorithm for the feature vector extraction is shown in Figure 5-4.

Algorithm: ExtractionofFeatureVector:
Input: Transactions
Output: Vectors

```

vectors = {}

map[attribute, features] = {}
For each transaction T in {transactions}:
    For each query Q in T:
        Parse Q by SQL parser;
        For each attribute A in Q:

```

```

        Identify feature for A;
    Endfor;
Endfor;
For each attribute A in T:
    If the operation is unique:
        map[A] += identified feature;
    Else
        map[A] += OU;
    Endif;
Endfor;
Endfor;
For each attribute A in map:
    Array vector[8] = {0};
    For i := 1 to 7:
        If there is ith feature for A:
            vector[i] = 1;
        Endif;
        If there is OU for A:
            vector[8] = 1;
        Endif;
    Endfor;
    vectors += vector;
Endfor. // Finished

```

Figure 5-4. The algorithm for feature vector extraction

5.2 Proposed Approach

The extracted feature vectors are filtered and passed to the clustering algorithm. We use the training data to estimate the optimal parameters values. Based on the clustering result and optimal parameters values, normal and anomalous clusters are identified. Then the model can be used to detect anomalies in a new database application. The overview of the approach can be seen in Figure 5-5.

Figure 5-5. Overview of the proposed approach

5.2.1 Clustering Method

The extracted feature vectors have a relatively low dimensionality and the elements of the vector are binary value. Single-linkage clustering is a type of hierarchical clustering by

grouping clusters from bottom-up. During the grouping process, the sample points which are similar based on distance metric would be aggregated together, while leaving the outliers alone. Based on this, single-linkage clustering method is well suited to deal with the data in our method. This algorithm works in the following steps: with a distance metric M and a constant W which is the width of the cluster, the distance between a cluster's defining instance C and a feature vector d can be computed as Euclidean $dist(C, d)$, where C is the centroid feature vector of that cluster. If the distance is smaller than W , the feature vector d will be assigned to the nearest cluster. Otherwise, a new cluster would be generated and this attribute feature vector would become its centroid. The Euclidean metric used to measure the distance between a cluster C and an instance d is defined as:

$$Dist(C, d) = \sqrt{\sum_{i=1}^n (c_i - d_i)^2}$$

where n is the dimension of the vectors, and c_i is the element of the centroid of C while d_i is the element of vector d .

5.2.2 Labeling Clusters

After clustering, it is still unknown that to which clusters the normal attributes belong and to which clusters the contain anomalies belong because during training the attributes are unlabeled. The assumption under our metric is that attributes with common features tend to be close together while those that have different features are sparse in between. If we have a cluster width W which is optimal, the clustering process would generate cluster set which can distinguish between normal and abnormal database attributes.

Since we chose industrial database applications to train our system, we assume that an overwhelming majority (>98%) of the training dataset is made up of normal attributes so that small clusters are likely to consist of anomalies. Hence, we can label the first $P\%$ of the clusters that contain higher number of attributes as 'normal', and the remaining clusters as 'anomalous'.

5.2.3 Detection of Anomalies

With the labeled clusters, we can proceed to conduct anomalies detection in usage of database attributes. For a new database application, we can extract the feature vector for each attribute. For each feature vector v extracted representing a database attribute, we find the cluster which includes v and label this attribute using that cluster's type. The detection proceeds as follows:

1. Find a cluster C which includes v under the metric M . Classify v according to the label of C .
2. If v does not belong to any clusters discovered thus far, a new cluster would be generated and v would become its center. Classify v as anomalous.

v is labeled as anomalous in the second step because of the assumption that, in a large-scale database application, it is generally complete enough to include all types of normal attribute. With such assumption, attributes that cannot be clustered according to the first step can be deemed as outliers. After the clustering, those attributes that are clustered into anomalous cluster would imply that the transactions involving them may contain anomalies.

5.3 Evaluation and Results

5.3.1 Training Dataset

Readily available dataset was extremely scarce in our research. Instead, we selected three matured, large-scale industrial database applications and seven open-source database applications from sourceforge.net based on the following criteria:

- The application size should be considerable, including both the number of transaction and attribute, in order to make the detection model meaningful.
- The applications have been used for some time and are mature so that they have very few database usage errors.

The three selected industrial database applications come from different domains. The first one is a membership management system to track members, families, groups, pledges and payments. The second one is a school management system which is used to manage course registration, staff and student information. The last one is a web-based e-commerce system which is used for product ordering. Furthermore, the seven open-source database applications also cover several domains. ChurchInfo is a free database application to help churches track members, families, groups, pledges and payments. Front Accounting is a professional web based powerful system for the entire ERP chain. CourseMS is a course management system which performs online registrations and provide websites with published course schedules and control over registrations. Gliding Booking System is a highly customized online system which allows members to register to fly with a gliding club. Cite CRM is a complete Customer Relations Management System with everything a small and medium business would need. Alumni Server is an open-source alumni system for universities and other organizations. Hotel Booking Portal is used for hotel booking which contains both a frontend and a backend.

Table 5-1 shows some statistics of each subject. Finally, we obtained a dataset including with 8909 source code files, 1393K lines of code and 3302 database attributes in total. All the experiments were conducted on a desktop PC with an Intel Core Duo 2.4GHz CPU and 4GB memory.

For data collection, we have built our extraction tool on top of phc [75], an open-source PHP compiler, which could facilitate parsing and the construction of intermediate representation for PHP code. Our tool performs data flow analysis on the paths of the inter-procedural CFG for each system, and extracts all the database transactions from them. After that, feature vectors for all the attributes in these ten systems are formed.

Table 5-1. Systems used in evaluation

System	Version	#PHP file	#LOC	# Attribute	Analysis time
System 1	4.2.12	2628	278623	612	15m18s
System 2	3.5.4	1986	351968	458	19m12s
System 3	4.0.3	2062	320821	528	21m36s
Church Info	1.2.12	964	192482	406	15m30s
Front Accounting	2.3.1	631	155945	626	9m36s
CourseMS	2.2.1	178	29024	195	1m43s
Gliding Booking System	N/A	109	25606	104	1m36s
Cite CRM	0.2.0	181	19952	112	1m24s
Alumni Server	1.0.7	140	13957	153	52s
Hotel Booking Portal	0.1	30	4833	108	22s

5.3.2 Performance Measurement

Two measures were computed over all labelled attributes to access the performance of our approach.

- Detection rate = the number of anomalous attributes detected by the system / the total number of anomalous attributes presented in the testing dataset;
- False positive rate = the total number of normal attributes that were wrongly classified as anomalous / the total number of normal attributes.

These two measures are useful and good indicators of classification performance. The detection rate measures the percentage of anomalies our approach is able to detect while the false positive rate measures how many incorrect classifications our approach makes. To calculate these values, access to labels of attributes in the data set is required. Based on the database schema, we extracted the attributes in the database. After analysing the program code and based on the attributes extracted, we can identify whether proper and adequate operations

have been conducted on each attribute and then labelled each attribute with “normal” or “anomalous”. We calculated these two measures over all labelled attributes.

5.3.3 Fixing Parameters

We make use of 30% of the dataset (990 attributes) as our training dataset. In our experience, if less than 30% of the dataset is selected as training data, the representativeness of the trained model would be compromised and not suitable for predicting. Besides this, the distribution of the training data should also be general enough in order to reflect original normal and abnormal attributes’ distribution. Our first assumption states that normal attributes should greatly exceed anomalous attributes. In the resulting training dataset, the number of normal attributes (96.36%) greatly exceeds that of the anomalous attributes (3.64%).

The values of two parameters should first be fixed and optimized: the cluster width W and the percentage P . W determines the minimum distance between two attribute feature vectors assigned to the same cluster, while the percentage P decides the top $P\%$ largest clusters that are labelled as ‘normal’.

The training dataset underwent a series of tests using a range of values of the two variables W and P . Table 5-2 shows the results for optimizing the values of W , and their corresponding measured performances. As we use the Euclidean distance measure, the cluster width W is set to a value slightly below each Euclidean distance value (such as $\sqrt{1}, \sqrt{2}, \dots, \sqrt{8}$).

Table 5-2. Statistics for evaluating w

W	P	DR	FPR	W	P	DR	FPR
0.7	60%	94.4%	0.419%	2.2	60%	58.3%	1.992%
1.4	60%	88.9%	1.048%	2.4	60%	47.2%	4.193%
1.7	60%	80.6%	1.468%	2.6	60%	38.9%	5.241%
1.9	60%	69.4%	1.572%	2.8	60%	33.3%	6.289%

(DR=Detection Rate; FPR=False Positive Rate)

The value of $W = 0.7$ was determined in subsequent tests because a low false positive rate and high detection rate was retrieved. Several tests on the same dataset were performed to find the value for P . The results of some of the tests are shown in Table 5-3.

Table 5-3. Statistics for evaluating p

W	P	DR	FPR	W	P	DR	FPR
0.7	80%	72.2%	0.314%	0.7	40%	97.2%	13.732%
0.7	70%	80.6%	0.419%	0.7	30%	97.2%	24.738%
0.7	60%	94.4%	0.524%	0.7	20%	100%	33.962%
0.7	50%	94.4%	6.499%	0.7	10%	100%	50.524%

From Table 5-3, it can be seen that the detection rates are the same when $P = 50\%$ or $P = 60\%$, though the false positive rate is fairly lower when $P = 60\%$. Therefore, P of 60% was chosen.

The clustering results of the training dataset when $P=60\%$, $W=0.7$ are shown in Table 5-4. There are 32 clusters in total. Based on the value of P , we labelled the first 19 clusters as normal and labelled the rest as anomalous. After further investigate of the anomalous attributes, we classified them into the following four types:

- Missing database operations (MI): For a database attribute, essential database operations (e.g., inserting a value of an attribute) are missing.
- Inconsistent database operations (IC): It is essential to provide a transaction in a database application to correct the effect of a transaction that has been executed with erroneous input. For a transaction that updates an attribute through cumulative update, the correction should also be made through cumulative update for control purpose. Correcting the result of a transaction that updates an attribute by “cumulative update” using “overriding update” is a common inconsistency fault.
- Redundant database operations (RD): For a database attribute, additional different types of database operations are performed on it.
- No Update (NU): For a database attribute, the program does not provide any operation to maintain or to use it.

It can be seen from Table 5-4 that there are a total of 956 attributes (96.57%) labeled as “normal” (NM) and 34 attributes (3.43%) labeled as “anomalous” (MI, IC, RD or NU).

Table 5-4. Clustering results of the training dataset

Cluster	Center of clusters	# of attribute	Type	Cluster	Center of clusters	# of attribute	Type
1	<1,0,0,1,0,1,1,0>	113	NM	17	<1,0,1,0,1,0,1,0>	26	NM
2	<1,1,0,1,0,1,1,0>	94	NM	18	<1,1,1,1,0,0,1,0>	25	NM
3	<1,0,1,0,0,1,1,0>	82	NM	19	<1,0,0,0,1,0,1,0>	23	NM
4	<1,0,1,1,0,1,1,0>	76	NM	20	<1,0,0,1,0,1,0,1>	5	RD
5	<1,1,1,0,0,1,1,0>	71	NM	21	<1,0,0,0,0,1,1,0>	4	NU
6	<1,0,1,0,1,1,1,0>	65	NM	22	<1,0,0,1,1,0,1,0>	4	IC
7	<1,1,0,0,1,1,1,0>	56	NM	23	<0,1,0,1,1,1,0,0>	3	IC
8	<1,1,1,1,0,1,1,0>	48	NM	24	<0,1,0,0,0,0,1,0>	3	NU
9	<1,1,1,0,0,0,1,0>	42	NM	25	<1,0,0,1,1,1,1,0>	3	IC
10	<1,1,1,0,1,1,1,0>	41	NM	26	<0,0,0,0,0,1,0,0>	2	MI
11	<1,1,1,0,1,0,1,0>	39	NM	27	<0,0,0,0,0,0,0,0>	2	MI
12	<1,0,0,1,1,0,1,0>	37	NM	28	<0,0,1,0,1,1,1,0>	2	MI
13	<1,0,0,1,1,1,1,0>	32	NM	29	<0,0,1,0,0,1,1,0>	2	MI
14	<1,0,1,1,0,0,1,0>	30	NM	30	<0,0,0,1,0,0,1,0>	2	MI
15	<1,1,1,1,0,1,0,0>	28	NM	31	<0,0,0,0,0,1,1,1>	1	MI
16	<1,1,0,0,1,0,1,0>	28	NM	32	<0,1,0,0,0,0,1,0>	1	NU
Total number of attributes: 990							

5.3.4 Cross Validation Testing

Based on the parameters and clusters learned from the training process, we performed cross-project evaluation by using a variant of the cross validation method.

We formed five subsets from the entire dataset, each containing approximately 660 attributes. It should be noted that no information about the attributes' label was used during the clustering in the training process; therefore, the training set still can be used for testing. Since we require that the number of anomalous attributes should constitute a very small portion of the training dataset, it was found that there were two subsets failed to meet this requirement. Therefore for cross validation training only three of the five subsets are selected. The numbers of normal and anomalous attributes of each of the three subsets are shown in Table 5-5.

The clustering process selected one of the three subsets as the training dataset each time, labeling the clusters and using the chosen subset to test each of the other two subsets. A total of 6 times were these tests executed, and the results are shown in Table 5-6.

The proposed approach was shown to be able to achieve a high detection rate with a low false positive by our evaluations. It is our belief that this technique is able to detect the anomalies in attribute usage in database applications.

Table 5-5. Statistics of the three subsets

Training Set	#Normal attribute	#Anomalous attribute
T1	641	19
T2	637	23
T3	635	25

Table 5-6. Performance of the system under various training and testing dataset combinations

Training dataset	Testing dataset	Detection Rate	False Positive Rate
T1	T2	91.3% (21/23)	0.63% (4/637)
T1	T3	84.0% (21/25)	0.78% (5/635)
T2	T1	89.5% (17/19)	0.62% (4/641)
T2	T3	92.0% (23/25)	0.63% (4/635)
T3	T1	100.0% (19/19)	0.47% (3/641)
T3	T2	100.0% (23/23)	0.31% (2/637)

5.3.5 Some Examples

In this subchapter, some examples of detected anomalies in database usage in open-source database applications have been presented.

The attribute “subject” in the table “as_newsletter” in AlumniServer is predicted as missing function. The attribute is actually referenced. However, there is neither INSERT nor UPDATE operation on it, and this attribute is declared as “default null” in the database schema. This could be a potential fault because null value frequently causes unexpected result and even failure of the system.

In the prediction results of ChurchInfo, we found several no update attributes in table “user_usr”. This table is used to store the information of users, including user ID, password, etc. There are eight attributes named “usr_CalNoSchool1”, “usr_CalNoSchool12”, ... , “usr_CalNoSchool8” in this table, all of which are “date” data type. However, only the attribute “usr_CalNoSchool1” among the eight has UPDATE operation. It is reasonable to argue that this could be inappropriate. After carefully searching the system, we spotted the following code snippet in the source code file Default.php, which updates the table “user_usr”:

```

$ssSQL = "UPDATE user_usr SET .....
if ($_SESSION['dCalNoSchool1'] != '')
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;
if ($_SESSION['dCalNoSchool2'] != '')
    //!wrong attribute name starting from here
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;
if ($_SESSION['dCalNoSchool3'] != '')
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;

.....
if ($_SESSION['dCalNoSchool7'] != '')
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;
if ($_SESSION['dCalNoSchool8'] != '')
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;

```

After investigation, we found that the developer in fact intended to update all these eight attributes according to some conditions. However, all the strings in the concatenation were likely mistyped as “usr_CalNoSchool1”, resulting in no update operation for the other seven attributes. Aided by the prediction results, developers or maintainers can take corresponding actions to deal with this.

We only gave a few examples here to show the use of the approach. However, we believe that these examples have already demonstrated the benefits of the applications of our proposed approach.

5.4 Discussions

5.4.1 Variations to Clustering and Detection

Other than evaluating performances based on the cluster width and the percentage of largest clusters labeled as normal, experiments have also been conducted on different clustering methods.

In one variation of the clustering technique, multiple passes are allowed for generating and assigning instances. The original method, which performs clustering in only one pass, has already been discussed in Chapter 5.2. The main disadvantage of this is that, during the training, attributes have fewer clusters with which to group themselves into. This means that an attribute that belongs to a new type may be erroneously clustered to the closest existing cluster because the cluster that it actually belongs to has not yet been created. To avoid this kind of premature sub-optimal clustering, our new clustering technique first generated clusters without assigning any attributes to them in the first pass, and then assigned attributes from the training dataset into the complete set of clusters in the second pass. From our tests, we found that this variation of clustering performed nearly as well as that of the original, with the false positive rate just slightly above that of the original. This could be because fewer attributes were now assigned to each cluster on average.

We also tried another clustering variant which assigns each attribute to the label attached to the majority of the instance’s N closest clusters. For example, an attribute that is surrounded by N clusters would be labeled as normal if the majority of those N clusters were labeled as normal, or else it would be classified as anomalous. Our tests indicated that this technique was on par with our proposed approach.

Other than clustering methods to detect anomaly, classification methods are ruled out for the requirement of labels of the training dataset. Other techniques, like rule-based anomaly detection, are not suitable for the problem we target because no specific prior knowledge can

be assumed at the risk of loss of generality in the detection. Some data mining techniques, such as association rules mining and time series mining, can be applied on other kinds of anomaly detection, but they are also not applicable to detect the database attribute usage anomalies.

5.4.2 Assessing Performance

The trade-off between the false positive and detection rates is existent for most machine learning techniques, and this is clearly demonstrated in our approach. When more clusters are labeled as anomalous due to the proportional decrease in number of largest clusters labeled as normal, detection rate surges up considerably. Any anomalous attribute assigned to clusters previously classified as normal would now be correctly classified as anomalous when the percentage of clusters labeled as normal decreases. However, this detection is conversely traded-off against false positive rate because more normal attributes are now classified as anomalous under clusters that become labeled as such. From our assumptions, anomalous clusters should not have any normal attributes assigned in the first place, but this false positive becomes more evident when the ratio of largest clusters labeled as normal decreases.

5.4.3 Threats to Validity

It can be argued that the values of the parameters we evaluated are domain-dependent. In order to prove our technique, database systems of a variety of sizes and complexities from different domains have been chosen. Ideally, we would like both of our assumptions mentioned in Chapter 5.1 to be satisfied. In reality, of course this assumption may not be fully satisfied and this is also one of the primary reasons that our method fails to detect 100% of the anomalous. However, we still believe that the proposed approach can be applied to a variety of database applications, and the best way to prove our conclusions is to replicate and extend our experiments.

5.5 Related Work

Data mining techniques have been utilized in software engineering fields because of their efficiency in discovering latent knowledge. Recently, data mining methods have been applied to anomaly detection for different types of applications. Applying unsupervised anomaly detection for network intrusion detection has attracted interests in the academic community. Portnoy [77] et al. applied clustering to identify intrusion by treating intrusion as anomaly against normal behaviors. Eskin, et al. [27] compared the effectiveness between the fixed-width clustering algorithm, an optimized version of the k-nearest neighbor clustering algorithm, and the one class support vector machine algorithm. Furthermore, Eskin et al. [26]

employed machine learning method to learn a mixture of probability distributions in order to model the anomalies for intrusion detection. Oldmeadow et al. [69] researched on the clustering methods mentioned in [27] and showed that accuracy can be enhanced by clusters adapting to changing traffic patterns. Supervised anomaly detection in network intrusion detection, which uses only normal instances as training data, has been widely researched on in the literature as demonstrated by the survey by Lazarevic et al. [49].

Detecting anomalies in data is not new, and has been studied as early as the 19th century. The survey by Chandola et al. [15] on the anomaly detection techniques includes classification-based and clustering-based methods, as well as the applications of such techniques. Among the clustering algorithms in the literature, there are multiple categories such as: partitioning methods, hierarchical methods, density-based methods and grid-based methods. Our work is closely related to partitioning method which constructs k partitions of a database of n objects where each partition represents a cluster. Based on the assumption that normal data instances belong to large and dense clusters, while anomalies either belong to small or sparse clusters, several approaches have been proposed [45; 52; 60; 79]. The aforementioned studies demonstrate that by mining the characteristics exhibited by most types of anomalies in programs, we can find the anomalies effectively. Anomalies in database operations can also be found by means of data mining methods. Fix width clustering is widely applied in unsupervised anomaly detection methods as it is a linear time approximation algorithm [69; 77]. The width can either be a user-specific parameter or can be derived from the data. In our method, we conduct the experiment by using several repetitions of the algorithm to choose the best value of w .

Classification algorithms have also been used for anomaly detection. Wei et al. [28] proposed a method to generate synthetic anomalies based on known classes to detect known and unknown network intrusions. Steinwart et al. [87] used a theoretical framework of density-level detection to create a comprehensive classification approach to anomaly detection. Classification technique trains the model based on labeled data which may not always be available. Hence, when there is only unlabeled data, heuristics for anomaly detection is usually used. A labeled dataset is generated by assigning one label to the original unlabeled data and another to a set of the artificially generated data, and then a binary classification algorithm is applied [28; 107]. In our approach, we also generate errors in mature and correct systems to train our model. The labels “normal” and “anomalous” are assigned to the original unlabeled data and the artificially generated anomalies respectively. These labels are then used to evaluate the optimal system parameters W and P .

Methods other than clustering and classification have also been put forward for anomaly detection. For example, hidden semi-Markov model was used by Xie et al. [106] for the

detection network DDOS. Vigna et al. [98] combined analyses on HTTP requests and SQL queries to detect web-based attacks by. Zhang et al. [109] calculated the anomaly score of a data instance as the sum of its distances from k -nearest neighbors. These works are different from ours because focus is made only on the syntactic level of the queries but not system semantic level. Our work shows that anomalies in database operations can be discovered through data mining methods.

Over the years, several researches have been performed to explore automatically fault detection. Several researchers have used coverage based information for fault localization. For example, Jone et al. [39] proposed the Tarantula method for ranking suspicious statements based on program execution information. They also visualize coverage information to assist the process of fault localization. Renieris et al. [78] proposed a nearest-neighbor based algorithm to identify suspicious regions in the program source code by using similar program spectra. Their approach selects the correct run that most resembles the faulty run based on a distance criterion. Fatta et al. [24] proposed to mine frequent subtrees from the function call trees of test traces. The fault probability of a frequent subtree is calculated as the proportion of the subtree's appearances in the failed tests to the appearances in all tests. Dickinson et al. [25] checked for the program failures by clustering program execution profiles, in addition to proposing a technique to cluster failure reports within the space of selected features automatically [76]. In their method, they adopted both the supervised and unsupervised pattern classification and multivariate visualization.

Different from the aforementioned approaches, our approach is to mine anomalies in database attribute usage from source code using unlabeled feature vectors. Our approach is able to efficiently detect anomalies in database applications. We have applied the proposed approach to three large-scale industrial database applications as described in Chapter 5. These applications contain around 5000 source files and more than 540KLOC. Hence, we believe our approach can be applied to large-scale database applications. Furthermore, our approach can be used in the verification and validation of database applications. To date, we believe that no such approach has been proposed before. In opposition to our earlier work on data lifecycle [58], this thesis applies data mining techniques to systematically detect anomalies in database attribute usage, instead of using heuristic rules.

5.6 Conclusion

In this chapter, we have presented an approach for detecting anomaly in the use of database attributes based on feature vectors extracted from database applications. The proposed approach is able to detect the anomalies of attribute usage for database applications with high accuracy while keeping the false positive rate reasonably low. On average, the detection rate is

92.8%, and the false positive rate is 0.57%. The proposed approach can be used in the verification and validation of database applications.

Chapter 6

ATTRIBUTE LIFECYCLE COVERAGE TESTING OF DATABASE APPLICATIONS

Database makes up an essential part of many modern software applications, and even today, database applications are inherent in a major portion of software systems in the industry. It is therefore important to ensure that their behaviors are error-free.

According to a study by the NIST, software faults alone contribute up to a total of 59.5 billion US dollars, but more interestingly, over one-third of these faults occur during the development process. It can therefore be seen that testing of software products is indispensable but extremely costly in the development and maintenance process. It is estimated that half of the expenses caused by inadequate testing could have been avoided if proper testing infrastructure has been provided.

However, relatively little research effort has focused on how to test database applications. Database applications exhibit very specific characteristics, so it requires specific testing methods to test these specific characteristics. However, database applications are not just written in a purely imperative language like C programming language, but are usually consisting of a combination of an imperative language and a declarative language such as embedded SQL. As such, existing software testing methods which are designed for imperative languages cannot be directly applied to database application programs. Additionally, the information contained in the database compounds the difficulty in designing tests.

The above-mentioned reasons clearly justify the need for new approaches and tools support for testing database applications. This chapter is to address this issue.

Database typically maintains persistent data which experience a lifecycle. In a database record, for each attribute, a value is created initially. Then, the value can be referenced, or updated to a new value. Referencing and updating can occur in any order. Eventually, when the record is deleted, the value of the attribute is deleted together with it. From a general abstraction, all functionalities provided in a database application are for the purpose of transiting its attribute values to appropriate lifecycle stages.

Traditional testing methods usually focus on improving the path coverage of the source code. However, path coverage does not guarantee proper testing quality for the database applications. The following PHP code is an example.

```
<?php
$query = "UPDATE users SET username = " .
$_POST['usertype'] . " WHERE userid = " .
$_POST['userid'];
$result = mysql_query();
if (!$result) {
    die('Invalid query: ' . mysql_error());
}
?>
```

From the above code, it can be seen that there is no conditional guard in the program against the SQL query, which means the path can be deterministically executed by a test case. However, the SQL query embedded in the query is an UPDATE operation, which involves a conditional judgment against the user input. A test case may not trigger the update in the database, which means the test case cannot cover the attributes' lifecycle. Hence, in the testing of a database, it is crucial to exercise all the attribute lifecycle stages in a database application. In this chapter, we propose a database lifecycle testing method as well as a coverage measurement to guide the testing process.

We summarize our contributions as follows:

- We present a testing method for the design of test cases from white-box strategies.
- We propose a test coverage analysis to measure the quality of a test suite.
- We implement a tool to automate the measurement of the coverage of attribute lifecycles.

This chapter is organized as follows. Chapter 6.1 presents the attribute lifecycle coverage testing. Chapter 6.2 introduces how we extract the conditions in the database applications. Chapter 6.3 proposes a test coverage analysis to measure the quality of a test suite. Subsequently, the evaluation results run on real database applications are described in Chapter 6.4. Chapter 6.5 presents the related work. Chapter 6.6 concludes the chapter.

6.1 Attribute Lifecycle Coverage Testing

The core and most important part of testing a software system is the design of a set of test cases, called a test suite which is used to exercise the system as far as possible with the objective to reveal the maximal errors in the system. The design of a good test suite is the key for testing any system.

In general, testing techniques can be classified into black-box and white-box testing. Black-box testing designs test cases based on system specification. White-box testing designs test

cases based on the structure and logic of code. Many black-box and white-box testing techniques have been proposed [5; 110] with the objective of providing coverage for different aspects in specification and code. Our attribute lifecycles coverage testing can be applied based on code. We call it attribute lifecycles coverage testing and shall discuss this technique.

Let us take a simple example: a small database managing user account. In this application, there would be a database schema in SQL with one table. Table 'cus' holds data about customers of a bank.

```
CREATE TABLE cus (cusno INT PRIMARY KEY, cusname CHAR (25) UNIQUE NOT NULL, passwd  
VARCHAR (32), last_login DATE, amount MONEY, CHECK (amount>50));
```

As we can see from the schema definition, there are five attributes in this table. There should be a test case that executes the account registration function so that the creation stage of these attributes can be tested. Similarly, there should also be a test case to check the account deletion function. Again according to the specification, a user can modify his / her name and the password, and the last login time should be updated whenever the user logs in to the system. Therefore, there should be update operation for all the attributes except the attribute 'cusno'. All these specifications require corresponding test cases which form a test suite that could cover the attribute lifecycles of all these five attributes. Based on the code of database applications, attribute lifecycle coverage testing creates a test suite to transit at least one value of each attribute to cover all the attributes' lifecycle stages.

For the design of attribute lifecycle coverage testing, first, we extract a set of all the paths that go through any loop at most one time from the CFGs of programs in a database application. Next, we select a subset of paths from the set such that for each attribute and each following database operation performed in the database application there is at least one path which passes through a SQL statement. Different operations indicate that the specific path fulfills different lifecycle stage for the involved attributes.

- Insert: The involved attributes would have the insertion lifecycle stage, which means a record of the attributes is created. This is the beginning of a database attribute life cycle. If an attribute misses this stage, it should never be used afterwards or else logical fault is implied.
- Select: An attribute value should eventually be used or otherwise it has no meaning of existence. If an attribute has never been through this lifecycle stage, it indicates redundancy of the database schema design.
- Update: Although an attribute does not necessarily need to have this stage, update is a common operation for most of the attributes in database applications. The applications

use update to reflect business logic hence the completeness of the functionality of a system relies on this stage.

- Delete: This is also an optional stage for an attribute. However, missing this stage may still indicate incompleteness of the business logic specification. Whether this is a problem needs further inspection from the developers.

Since substantial improvement has been made in constraint solver in recent year, with the use of constraint solver, automated test case generator can be developed to generate the test suite. With the generated test suites, the coverage can be calculated automatically.

However, the required lifecycles of different attributes would be different. For example, for a content management system (CMS), the administration account may be initialized after installation, and cannot be deleted by any program. Hence, the related database attributes would not have delete lifecycle.

6.2 Extraction of Conditions of database applications

For database applications, the execution of a SQL query is not only depended on the program control flow, but also on the SQL conditions in the WHERE clause. Hence, traditional test case generation methods are not suitable for testing database applications. In our approach, we propose to use the combination of both program control flow condition expressions as well as SQL execution conditions. To do so, we need to extract conditions inside SQL queries aside from program control conditions.

6.2.1 SQL Extraction

In order to obtain the data to characterize attribute lifecycle, SQL queries in a database application need to be extracted. The SQL queries in the source code of database applications are commonly formed by concatenating several string literals and variables. Some parts of the query string can vary during execution. The following PHP code snippet provides an example.

```
<?php
function exec_query($q)
{
    return mysql_query($q);
}

$query = "SELECT username FROM users
        WHERE ";
if (isset($_POST['usertype']))
{
    $query .= "usertype = " .
        $_POST['usertype'];//use usertype
}
else
{

```

```

$query .= "userid=" .
    $_POST['userid']; //use userid
}
exec_query($query);
?>

```

In the above example, the query can be different in runtime. Moreover, the query is passed to another function `exec_query` for execution. In PHP, those lines of code that are not within a function scope can be viewed as in one virtual MAIN function. Regarding this, in order to enhance the completeness of query extraction, we employ a path-sensitive and context-sensitive inter-procedural data flow analysis.

Figure 6-1 shows the inter-procedural CFG for the above code snippet. In Figure 6-1, the box on top represents the control flow of the virtual MAIN function while the bottom box represents that of the “`exec_query`” function. The solid and dashed lines represent intra-procedural and inter-procedural control flows respectively.

Figure 6-1. Inter-procedural CFG for PHP code snippet

By following the data flow paths, we could replace variables used in the SQL queries with their original values.

6.2.2 Extraction of Conditions of Database Applications

- *Equality.*

The condition is to decide the equality or inequality between the value of a table column and a provided value from the program. For example, ‘WHERE username=\$username’. We extract the condition as a prefix expression: (eq COL(username) VAR(\$username)), in which ‘eq’ denotes ‘equality’ relation, COL indicates a column name, and VAR indicates that this is a variable name from the program.

- *Range.*

Range condition refers to the range operator like greater than or less than. For such cases, the extracted condition is like: (GT COL VAR). The operators include: GT: greater than; LT: less than; GTE: greater than or equal to; LTE: less than or equal to.

- *Containment.*

This refers to the set subsuming operator. For example, ‘WHERE status in (1, 2, 3)’, in which a set is provided to determine whether the column value is contained in it. This kind of condition is hard to be expressed in an expression since the set is usually unknown statically. We record the participating column name as well as the set, either a variable or a known set, in a pair for future reference.

- *Logical connective.*

Logical connective operators including AND, OR and Not are used to connect several conditions. Hence the extracted expression which logical connectives are involved would be essentially an expression tree, with each component itself also being an expression. For a WHERE clause ‘WHERE username=\$username AND date<’20140501’ and id>=100’, the extracted conditional expression tree would be:

```
(AND (AND
      (EQ COL(username)
        VAR($username))
      (LT COL(date) '20140501'))
      (GTE COL(id) 100))
```

For each SQL query that has the WHERE clause, the conditional expression is extracted for use of generating test case input data and database states.

6.2.3 Program condition expression extraction

Besides the conditions inside the SQL queries, we also need the conditions in the program. By traversing the program control flow, we identify the control nodes and extract the condition expressions from them. Note that such expressions involve only the variables or constants from the program source code.

- *Generation of Test Input*

With the extracted condition expressions from the program source code, we can generate test input for executing each path. This is also done in traditional path coverage testing. To achieve this, for each path, an SMT constraint solver is employed to solve the combination of the path conditions. The solution provided by the constraint solver becomes a test input set to cover this specific path.

- *Generation of Database States*

To guarantee the coverage of SQL queries, the program path coverage is not sufficient enough since even though the program path is executed, the embedded SQL query may still not be effective due to the WHERE clause evaluation. Therefore, we need to produce proper database states that work corporately with the program inputs so that the SQL query can actually take effect.

With the generated program input for one path, the values of variables which are used in one SQL query are known. Hence the variables in the extracted conditions from the SQL queries are only the column values, i.e. the COL parts as mentioned above. The SMT constraint solver is once again applied to solve the condition expression for each SQL query that has WHERE clause. The solution from the solver would be the value set that need to be set for the corresponding column to the database.

6.3 Test Coverage Analysis

Test coverage analysis is important to measure the quality of a test suite. Based on the criterion of attribute lifecycle coverage, we propose a new test coverage measurement, called attribute lifecycle coverage to measure the quality of any test suite for a database application. For a test suite T to test a database application, the attribute lifecycle coverage of T, ALC (T), is defined as follows:

$$ALC(T) = \frac{\text{Total number of attribute lifecycle stages exercised by } T}{\text{Total number of all the attribute lifecycle stages}} \times 100$$

Each attributes has at most four lifecycle stages (inserted, used, updated and deleted) supported in a database application. Most attributes should have all the four stages.

6.4 Evaluation

6.4.1 Dataset Description

We conducted evaluation for the proposed test case generation approach and the testing coverage criterion. We chose four open-source PHP systems from sourceforge.net for use in the evaluation. The four systems vary in source code quantities and the numbers of database attributes. Table 6-1 shows the statistics of the systems used in the evaluation.

Table 6-1. Statistics of systems in evaluation

System	Version	#PHP file	#LOC	# Attribute	Analyze time
batavi	1.0.6	4993	700321	1563	36m12s
webERP	4.0.2	594	167913	1083	10m36s
Front Accounting	2.3.22	366	76315	626	7m36s
OpenBusinessNetwork	2.0	120	10378	73	56s

6.4.2 Research Questions

In this evaluation, we intend to answer the following research questions:

RQ1: What is the effectiveness of the test case generation approach?

The efficiency of program fault detection is indicated by the effectiveness of the proposed test case generation approach. To answer this question, we measure the detection rate of the source code and SQL respectively.

RQ2: What is the effectiveness of the lifecycle coverage criterion?

Effectiveness of a coverage criterion indicates the ability of criterion to assess the quality of test suite. We explore the effectiveness of the traditional testing method on the testing systems and measured the detection rate, ALC etc. for demonstration.

RQ3: How does the proposed test case generation approach perform under the proposed coverage criterion?

Traditional test case generation methods were not specifically designed for database applications; therefore most of them cannot detect faults of the attribute lifecycle, and thus perform not well under the lifecycle coverage criterion. In this research question, we would like to answer how the proposed test case generation approach performs under the proposed coverage criterion and compared to traditional test case generation method.

6.4.3 Evaluation Results

To answer RQ1, we applied program fault injection to produce faulty programs. In this evaluation, we manually injected program faults into both the program source code and the SQL queries. Then test suites were generated and executed to find faults. We evaluate the effectiveness of the proposed approach by manually examining the injected and detected faults to determine the correctness of detection. The evaluation results are shown in Table 6-2.

Table 6-2. Results of evaluation for RQ1

System	#faults in source code	#faults Detected	Source code fault DR %	#faults in SQL	#faults in SQL Detected	SQL fault DR %
batavi	70	57	81.43%	80	52	65.00%
webERP	40	28	70.00%	50	37	74.00%
Front Accounting	45	29	64.44%	55	38	69.09%
OpenBusiness Network	20	13	65.00%	30	18	60.00%
Total	175	127	72.57%	215	145	67.44%

From Table 6-2, we can see that the generated test suites are capable of detecting the faults from both the program source code and the SQL queries. More specifically, the highest detection rate of faults in SQL queries is 74.0% and the overall detection rate of the two kinds of faults are 72.57% and 67.44% respectively. From the results, we can see that the proposed test case generation approach is capable of detecting program and SQL query faults effectively.

To answer RQ2, we deployed the traditional test case generation approach, which does not consider the conditions inside SQL queries when generating test inputs, to generate test suites for the open-source systems under testing. After applying the test suites, we measured the detection rate and the coverage criteria. For comparison, we measured the path coverage as well as ALC. The evaluation results are shown in Table 6-3.

From Table 6-3, we can see that although path coverage and ALC correlate with the detection rate, they are not directly relevant. For example, the statistics show that a high percentage of path coverage does not guarantee satisfactory ALC value. Moreover, the path coverage alone cannot effectively indicate the coverage of SQL fault detection. Therefore, ALC is required for use in the database application testing.

Table 6-3. Evaluation results of RQ2

System	Source code fault DR %	SQL fault DR %	Path Coverage	ALC
batavi	70.00%	51.25%	24.50%	19.70 %
webERP	57.50%	64.00%	32.30%	26.30 %
Front Accounting	48.89%	56.36%	57.25%	38.20 %
OpenBusinessNetwork	55.00%	43.33%	76.48%	50.10 %

Figure 6-2 shows the ALC plotted together with the percentages of paths for the four test suites. It can be observed that the increase of paths surpass the increase of ALC, which

indicates that having larger number of paths does not produce high ALC. This is because the number of paths does not guarantee coverage. For example, the selected paths may only include the SELECT operations for some attribute but fail to cover the INSERT and UPDATE. This demonstrates that employing attribute lifecycle coverage is important to measure the adequacy of database testing.

Figure 6-2. The ALC and the percentage of path

To answer RQ3, we compared two different kinds of test case generation methods. The first method only uses the path conditions in the source code, which does not consider the conditions inside the SQL queries. The second method is the test suite generation approach proposed in this chapter, which takes the SQL query condition into account for generating test input. These two kinds of methods were applied on the open-source systems under test and the ALC were measured for comparison. The evaluation results are shown in Table 6-4.

Table 6-4. ALC results of RQ3

System	Traditional method ALC	Proposed method ALC
batavi	19.7%	49.9%
webERP	26.3%	59.2%
Front Accounting	38.2%	63.1%
OpenBusinessNetwork	50.1%	62.4%

From the evaluation results in Table 6-4, it can be inferred that under the measurement of ALC, the proposed test suite generation method performed notably better than the traditional method. Besides, during our manual examination, it was discovered that the proposed method did better in uncovering faults which can only be exposed by combined program and SQL conditions.

6.4.4 Case study

Low attribute lifecycle coverage indicates the low quality of a test suite. We use an example to illustrate this. In this SUT, there is a source code file Default.php, from which a code snippet is shown below:

```
$ssSQL = "UPDATE user_usr SET .....  
if ($_SESSION['dCalNoSchool1'] != '')  
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;  
if ($_SESSION['dCalNoSchool2'] != '')  
    //!wrong attribute name starting from here  
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;  
if ($_SESSION['dCalNoSchool3'] != '')  
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;  
  
.....  
if ($_SESSION['dCalNoSchool7'] != '')  
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;  
if ($_SESSION['dCalNoSchool8'] != '')  
    $ssSQL .= ", usr_CalNoSchool1 = '". ...;
```

From the code, we can infer that the programmer intends to update different attributes according to different user input. However, all attribute names in the query have been mistakenly set to the same one, which results in other attributes not having update stage due to these code errors. For a test suite, even though it could cover all the program paths, it still cannot find the underlying bug. However, the ALC of the test suite would clearly indicate that several attributes lack of important lifecycle and the ALC would be low. Hence, low ALC value would suggest that there could be program errors.

6.5 Related Work

Since database becomes a crucial component for modern software, testing the database operations properly is important to ensure the quality. Therefore, research has been conducted on how to make the database testing more effective. For example, some works explore the mutation testing on database applications [94; 95; 111]. However, these methods have not taken full consideration of the coverage measurement.

Several researches in the literature try to address the problem with database test data generation. D. Chays et al. proposed AGENDA [16], a framework to populate the database states for testing with meaningful data. The tool could analyze the database schemas, application source and a set of heuristics to generate both database states and program inputs. The tool can also facilitate the output validation. In [23], the AGENDA framework was extended to enable the capability of testing database transactions with different states. D. Willmor et al. described in [105] an approach to intentionally generate the initial states of the database for the purpose of testing by analyzing the constraints from the specification.

Generally, these previous researches concentrate on the test case data preparation instead of test coverage evaluation. Their methods can be used as mutual complementary with our approach.

In addition, there have been several research works exploring the testing adequacy and coverage measurements for database applications. G. Kapfhammer et al. proposed a set of test adequacy criteria for database-driven applications [43]. They extend the classic data flow coverage to include the def-use relationships among program variables as well as database attributes from the database interaction CFGs. Their approach focuses on evaluating the data dependencies on the program control flows, rather than the completeness of database attributes as we do. M. J. Suárez-Cabal et al. proposed an SQL coverage measurement by using the coverage tree [89]. The coverage tree is generated for each SELECT query by parsing the conditions in the WHERE clause. This method only considers the SELECT queries, which is only part of the attribute lifecycle. However, our proposed approach takes the attribute lifecycle completeness into consideration, which reveals more intrinsic faults such as incomplete implementation of the business logic.

6.6 Conclusion

The goal of this chapter is to improve the testing of database applications. We propose an approach to test the coverage of attribute lifecycles. We consider both the program control flow and conditions in the SQL operations. We also propose a test coverage analysis to measure the quality of a test suite. A tool has been implemented for the method. Our evaluation results indicate that our approach is effective in guiding testing and detecting errors of database applications.

Chapter 7

CONCLUSION AND RECOMMENDATIONS

In this chapter, we summarize the contributions of this research. We also discuss limitations of our current work and suggest future work that would enhance the solutions provided in this thesis.

7.1 Conclusion

Database makes up a large part of many software systems. The reliability in database applications is more complex than other software systems. Customers of database applications want more reliability during the use of those applications. So the reliability of the database applications is considered as a special case over different kinds of network.

There are many different approaches to detect and isolate faults and each approach has its benefits and limitations. The most popular and commonly used techniques include model based reasoning, fault pattern recognition, neural networks and hybrid approaches. We identified that these existing approaches suffer from one or more of the following weaknesses, which prevent them from comprehensively addressing the fault detection of database applications.

- 1) Previously, program analysis has been applied on software systems trying to find potential faults and bugs. Such approaches have also been carried out on database systems either on program source code or on SQL queries. However, most of the researches do not take the interaction between program source code and the database SQL queries into consideration. Such interaction is actually the key component of database applications because they constitute the most of the business logic. Furthermore, database systems may also have vulnerabilities which are caused by the unsafe control or data flow between program constructs and database operations.
- 2) Some efforts have been put on how to maintain the DBMS constraints enforcement efficiently and effectively. Although most database management systems can automatically enforce constraints by defining them in the database schema, upon which DBMS will reject any updating to a database that will lead to constraint violation. However, exception handling to handle such rejections still requires coding

by programmers. No research has explored the automatic exception handling for the violations of DBMS constraints.

- 3) Much research has been delved into exploring software change impact analysis, but little work has been done on improving the impact analysis of database schema. Current methods either only consider the individual database queries or analyze the program flow dependencies. However, data flow also plays an important role on impact analysis of database applications.
- 4) Classification techniques have been applied for anomaly detection for a long time, but a large quantity of training data is usually needed for effective classification. Generating these training data manually is difficult and time-consuming. In addition, a current notion of normal behavior might not be enough in the future because normal behavior keeps evolving. For these reasons, classification-based anomaly detection algorithms, which rely on labeled data, are often inaccurate and highly expensive.
- 5) Some approaches might be comprehensive and might be very promising in addressing testing or fault detection of database applications. But, they might fail in providing adequate implementations either commercially or publicly. From our experiences, it requires tremendous effort and knowledge in implementing the proposed theoretical concepts into practical tools. It is difficult for developers who have little knowledge in web based research to implement and adopt these approaches.
- 6) Traditional testing methods usually focus on improving the path coverage of the source code. However, path coverage does not guarantee proper testing quality for the database applications. Database applications are typically written in a variety of imperative language and declarative languages, rather than using a purely imperative language. Hence, current testing tools can hardly abide the nature of database applications.

Hence, based on these observations, in this thesis, we presented four novel approaches for analyzing database applications in an attempt to address the shortcomings of these existing approaches. Our work presented in this thesis has been published as listed in Section Author's Publications.

Program analysis plays an important role in most software engineering activities such as testing, debugging, reverse engineering, and program comprehension. It can be used to extract properties or characteristics of functions and statements in a program. On the other hand, data mining techniques have become extremely useful in software engineering tasks such as

predicting software quality. In this thesis, we have addressed the fault detection of database applications through the use of program analysis techniques and data mining techniques.

We believe that the faults of database applications may happen on different stages of the software lifecycle. For example, the database schemas designed by the developers initially might be inconsistent or redundant. During the implementation, the programmers may not handle all types of exceptions properly. Furthermore, when the database is modified later, if the associated impact has not been completely understood, the data would potentially be inconsistent, this may ultimately cause faults. Based on this, we suggest ways to detect and handle the faults from multiple angles. The four proposed approaches themselves can be complementary solutions to one another because the approaches address the issue in four different angles:

Database constraints exception handling: We first proposed an extended AST to include SQL query semantics based on traditional AST. Based on it, each code pattern that requires exception handling together with the exception handling code to be inserted was represented as a transformation rule. We provided two alternatives to handle the possible exceptions associated with key and referential constraints: one is to handle the exceptions in conjunction with the built-in enforcement feature in Database Management System; the other is handling them without using the feature provided in Database Management System. Hence, two types of transformation rules were provided accordingly.

A tool called *GEHPHP* has been developed to implement the proposed approach. Our tool uses *phc*, an open-source tool for parsing and compiling PHP programs to generate CFGs and perform control and data dependency analysis. Using the tool, a study was conducted to evaluate our pattern-based approach. For both ways of handling exceptions, our tool automatically generated and inserted the exception handling code for all the SQL queries that required exception handling. The correctness of the inserted code was manually examined in runtime by triggering the execution of exception handling code. As expected, all the exceptions were handled appropriately by our inserted code, and the error messages were printed correctly, which meant the managed portion of the exceptions was 100%. Database developers can make use of the proposed approaches to handle the exception handling of database applications. This can avoid human error in coding exception handling and prevent omission or inconsistent action when handling the same type of exception.

Database maintenance: We proposed a novel graph called the attribute dependency graph to reveal the dependencies between attributes in a database application and also the programs involved. We proposed an approach to automatically extract the attribute dependency graph out of a database application from its source code through inter-procedural static program

analysis. The extracted information is a straightforward indicator that benefits the maintenance process, particularly for impact analysis on the modifications in a database application. By using the attribute dependency graph, for any change that has been made to a database attribute, the developers are able to identify and investigate the change's impact to other attributes that depend on the changed attribute and respond accordingly with necessary actions.

A tool has been developed to implement the proposed approach. Case studies with three PHP systems have also been conducted to demonstrate the use of our approach. These case studies examined the impact of changes made to these database systems, including dropping of attributes, change to attributes and changes to programs. From the case studies, we could see that the proposed attribute dependency graph can assist the maintenance process by providing accurate visual description of attribute dependencies. As impact analysis of modification in database applications is time consuming, the attribute dependency graph can lighten the burden of the programmer and improve the efficiency of maintenance. In addition, the approach can avoid human error in examining and identifying of all the potential impact. Developers and business logic analyzers can utilize the proposed approach and tool to aid business and software system development and evolution.

Database attribute usage anomaly detection: We abstracted and characterized database operations performed on a database attribute by a feature vector extracted from code. A clustering-based anomaly detection approach which takes as inputs a set of unlabeled database attributes was proposed to detect anomaly in the usage of database attributes. Two general assumptions about the data have been made in our algorithm. The first one is that the number of normal attributes is much greater than that of the anomalous attributes, meaning that larger clusters should be formed by normal attributes than the anomalous attributes. The second assumption is that the anomalous attributes and the normal attributes are qualitatively different, thus implying that it is impossible for them to be grouped into the same clusters. Any anomalous attributes in the dataset would therefore appear to be outliers due to their rarity and abnormality.

Our approach grouped the database attributes together into clusters using a distance-based metric. Once the database attributes were clustered, we identified small clusters and labelled them as anomalous clusters. Then this model could be used for detecting anomaly for a new database application.

A prototype tool has been developed to implement the proposed approach. Using the tool, we conducted comprehensive experiments on a set of industrial and open-source PHP applications to evaluate the accuracy of our proposed approach. The results show that our approach is able to detect many types of anomalies with an average detection rate 92.8% and

false positive rate 0.57%. Results suggest that the proposed method can be efficient and effective, hence, providing a good solution to detect database faults. For database application developers, it is a key point to detect and identify potential design or implementation flaws as early as possible. The proposed approach is therefore useful to aid such process.

Database testing: Attribute lifecycles represent a general abstraction on the functionality of a database application. Hence, exercising lifecycle stages in testing database applications reflects testing of the functionality in general. Based on this intuition, we proposed a testing method to test the coverage of attribute lifecycles. We also proposed a test coverage analysis to measure the quality of a test suite.

A tool has been implemented for the method. Our evaluation results indicate that our approach is effective in guiding testing and detecting errors of database applications. The generated test suites are capable of detecting the faults from both program source-code as well as SQL queries. The overall detection rates of the two kinds of faults are 72.6% and 67.4% correspondingly. In addition, under the measurement of attribute lifecycle coverage, the proposed test suite generation method performed notably better than the traditional method. Testing is common process of a complete software development cycle. For database applications, it would be instructive if the testing process can feed back the database attribute lifecycle completeness and test coverage information.

7.2 Recommendations

In this subchapter, we present some research directions and future work which could improve the current research results demonstrated in this thesis.

In this thesis, we evaluate the proposed approaches with several experiments to prove the effectiveness. Although the experiments are conducted on different systems which differ in application fields, scale and design patterns, the test subjects might not be diversified enough to represent all types of systems. Besides, most of the experiments are carried out on web applications which are written in PHP even though our proposed approaches are generic without targeting at any specific system types. The reason for this is that such systems represent a large number of database systems for real world business. In the future, we plan to explore more heterogeneous real-world applications with our proposed approaches.

Exception handling for database systems is crucial to ensure that the business would not be interrupted by any abnormal transactions on the backend database, and unexpected exception would not interfere with the stability of the systems and the consistency of data. It is not uncommon that developers of database applications ignore or forget to implement proper exception handling mechanism, thus it is meaningful to provide a method to insert exception

handling code automatically. However, there is still database related logic which does not reside in program source code, such as the database storage procedures. For these parts, automatic exception handling could also be useful.

Program impact analysis and maintenance are two aspects which are explored from several different angles. One category is to use program analysis, to extract underlying knowledge from program source code and further infer the defects from it. Further challenging comes from the complicated interaction between program source code and database queries. We made preliminary attempt to relate the two parts of a database application to provide useful diagnostic information in the form of attribute dependency graph. The graph itself can be expanded so as to be semantically richer and can be used to express more complex relationships among database query elements and program constructs.

Testing of software systems is an intensive research area. For white-box testing, program source code has been explored with several analysis techniques, including static program analysis; symbolic execution and satisfiability modulo theories in order to generate test input suite and system states. Many tools have been developed targeting different kinds of applications. For database applications, there are also such research works, e.g. generate input combinations to test for SQL injection vulnerabilities. However, there are still few efforts devoted into incorporating program source code and database queries together to produce test cases which are more suitable for database applications. We perform an exploratory research on such combination, and evaluate the method with a newly proposed coverage criterion which covers the lifecycle completeness of a test suite. Further research direction can be to extend the approach to accommodate black-box testing as well as dynamic program analysis so as to leverage more knowledge from runtime.

Security issues are increasingly severe for modern day software systems. Malware intrusion, SQL injection (SQLI) and cross-site scripting (XSS) are common kinds of security threats. Traditionally, detection of such security vulnerabilities requires intensive and laborious manual inspection or testing. This is not only inefficient but also error-prone. High risk vulnerability may remain unfixed after release, and could cause serious loss upon breach. Automatic static and dynamic program analysis and data mining techniques can help mitigate such problems. For example, anomaly detection can be used to detect intrusion of web-based systems by training models based on database access patterns. SQLI and XSS vulnerabilities can also be detected by utilizing both static and dynamic code attributes. We are interested in applying and extending the proposed program analysis and anomaly detection approaches in this paper to security vulnerability detection problems accompanied with database and web applications.

Database tables and attributes are defined in the database schemas. Therefore, database schema provides important information on attribute's type as well as primary key and foreign key relationships. Besides, data access layer (DAL) is also a thin abstraction of raw database operations. Although database schema and DAL provide basic information, detail business logic and program flow are still only available from source code. Hence static program analysis combined with data mining is of significance to offer insights of the database applications. We believe that such analysis can be aided with knowledge of the database schema as well as the data access layer.

APPENDIX A

Table Appendix-1 Notations used

Symbol	Meaning
DBMS	database management system
SQL	structured query language
KRC	key and referential constraints
AST	abstract syntax tree
PK	primary key
FK	foreign key
CFG	control flow graph
ICFG	interprocedural control flow graph
DDG	data dependency graph
DB-AST	database abstract syntax tree
LOC	lines of code
SQL statement	statement in the program that creates, reads, or modifies records in the database
HTTP	Hyper Text Transport Protocol
ORM	object relational mapping
CMS	content management system
ALC	attribute lifecycle coverage

AUTHOR'S PUBLICATIONS

- [1] K.P. Liu and H.B.K. Tan, "Detecting anomaly in the usage of database attribute" in *Proceedings of 26th International Conference on Software Engineering & Knowledge Engineering*, pp. 705-710, July 2014.
- [2] K.P. Liu and H.B.K. Tan, "Faceted bug report search with topic model", in *Proceedings of 38th Annual International Computers, Software & Applications Conference*, pp.123-128, July 2014.
- [3] K.P. Liu and H.B.K. Tan, "Mining attribute lifecycle to predict faults and incompleteness in database applications" in *Proceedings of 20th Asia-Pacific Software Engineering Conference*, pp. 223-230, December 2013.
- [4] K.P. Liu, H.B.K. Tan and H.Y. Zhang, "Has this bug been reported?" in *Proceedings of 20th Working Conference on Reverse Engineering*, pp. 91-96, December 2012.
- [5] K.P. Liu, H.B.K. Tan and X. Chen, "Supporting the adaptation of open-source database applications through extracting data lifecycles", *IET Software*, vol.7, no.4, pp. 213-221, Aug 2013.
- [6] K.P. Liu, H.B.K. Tan and X. Chen, "Aiding maintenance of database applications through extracting attribute dependency graph", *Journal of Database Management*, vol.24, no.1, pp.20-35, 2013.
- [7] K.P. Liu, H.B.K. Tan and X. Chen, "Automated insertion of exception handling for key and referential constraints", *Journal of Database Management*, vol.24, no.1, pp.1-19, 2013.
- [8] K.P. Liu, H.B.K. Tan and X. Chen, "Binary Code Analysis", *IEEE Computer*, vol.46, no.8, pp.60-68, Aug.2013.
- [9] K.P. Liu and H.B.K. Tan, "Automated insertion of exception handling for key and referential constraints, " in *Proceedings of 19th Asia-Pacific Software Engineering Conference*, pp. 848-856, December 2012.
- [10] K.P. Liu, H.B.K. Tan and L.K. Shar. "Semi-automated verification of defense against SQL injection in web applications," in *Proceedings of 19th Asia-Pacific Software Engineering Conference*, pp. 91-96, December 2012.
- [11] K.P. Liu and H.B.K. Tan, "Has this bug been reported?" in *Proceedings of 20th International Symposium on the Foundations of Software Engineering*, 28:1-28:4, November 2012 (**top conference**).

- [12] K.P. Liu, H.B.K. Tan and X. Chen, “Extraction of attribute dependency graph from database applications”, in *Proceedings of 18th Asia-Pacific Software Engineering Conference*, pp. 138-145, December 2011.
- [13] K.P. Liu, H.B.K. Tan, X. Chen, H.Y. Zhang and B. Padmanabhuni, “Automated extraction of data lifecycle support from database applications,” in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, pp. 432-437, July 2011.

BIBLIOGRAPHY

- [1] AGRAWAL, R. and SRIKANT, R., 1994. Fast algorithms for mining association rules. *in Proceedings of the 20th International Conference of Very Large Data Bases, VLDB*, 487-499.
- [2] AHO, A.V., 2003. Compilers: Principles, Techniques and Tools (for Anna University), 2/e. Pearson Education India.
- [3] AMBLER, S., 2006. Relational databases 101: Looking at the whole picture. *Techniques for Successful Evolutionary/Agile Database Development*.
- [4] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., and ERNST, M.D., 2008. Finding bugs in dynamic web applications. *In Proceedings of the 2008 International Symposium on Software Testing and Analysis ACM*, 261-272.
- [5] BAU, J., BURSZEIN, E., GUPTA, D., and MITCHELL, J., 2010. State of the art: Automated black-box web application vulnerability testing. *In Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2010, 332-345.
- [6] BERZTISS, A. and THALHEIM, B., 2007. Exceptions in Information Systems, *In Proceedings of the 9th Russian Conference on Digital Libraries, RCDL '2007*.
- [7] BETHESDA, M. The SANS Institute. SQL Injection Prevention Cheat Sheet.(2012, July 9). OWASP The Open Web.
- [8] BLAHA, M., 2004. A Copper Bullet for Software Quality Improvement. *IEEE Computer*, 37, 2, 21-25.
- [9] BOHNER, S.A., 1996. Software change impact analysis.
- [10] BOND, M.D. and MCKINLEY, K., 2005. Practical path profiling for dynamic optimizers. *In Proceedings of IEEE International Symposium on Code Generation and Optimization*, 2005, 205-216.
- [11] BRAVENBOER, M., KALLEBERG, K.T., VERMAAS, R., and VISSER, E., 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1, 52-70.
- [12] CAROPRESE, L., GRECO, S., and ZUMPANO, E., 2009. Active Integrity Constraints for Database Consistency Maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 1042-1058.
- [13] CASTRO, P., MELNIK, S., and ADYA, A., 2007. ADO. NET entity framework: raising the level of abstraction in data programming. *In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM*, 1070-1072.

- [14] CHAN, H., SIAU, K., and WEI, K.-K., 1997. The effect of data model, system and task characteristics on user query performance. *ACM SIGMIS Database* 29, 1, 31-49.
- [15] CHANDOLA, V., BANERJEE, A., and KUMAR, V., 2009. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)* 41, 3, 15.
- [16] CHAYS, D., DENG, Y., FRANKL, P.G., DAN, S., VOKOLOS, F.I., and WEYUKER, E.J., 2004. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability* 14, 1, 17-44.
- [17] CLEVE, A., 2010. Program analysis and transformation for data-intensive system evolution. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*, 2010, 1-6.
- [18] CLEVE, A., HENRARD, J., and HAINAUT, J.-L., 2006. Data reverse engineering using system dependency graphs. In *Proceedings of Working Conference on Reverse Engineering, 2006, IEEE*, 157-166.
- [19] CORNELISSEN, B., ZAIDMAN, A., VAN DEURSEN, A., MOONEN, L., and KOSCHKE, R., 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35, 5, 684-702.
- [20] DASGUPTA, A., NARASAYYA, V., and SYAMALA, M., 2009. A Static Analysis Framework for Database Applications. In *Proceedings of IEEE 25th International Conference on Data Engineering, 2009. ICDE'09*, 1403-1414.
- [21] DAVE, C., BAE, H., MIN, S.J., LEE, S., EIGENMANN, R., and MIDKIFF, S., 2009. Cetus: A source-to-source compiler infrastructure for multicores. **Computer** 42, 12, 36-42.
- [22] DE STEFANO, C., SANSONE, C., and VENTO, M., 2000. To reject or not to reject: that is the question-an answer in case of neural classifiers. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 30, 1, 84-94.
- [23] DENG, Y., FRANKL, P., and CHAYS, D., 2005. Testing database transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering ACM*, 78-87.
- [24] DI FATTA, G., LEUE, S., and STEGANTOVA, E., 2006. Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd International Workshop on Software Quality Assurance ACM*, 62-69.
- [25] DICKINSON, W., LEON, D., and PODGURSKI, A., 2001. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering IEEE Computer Society*, 339-348.

- [26] ESKIN, E., Anomaly Detection over Noisy Data using Learned Probability Distributions. *In Proceedings of the International Conference on Machine Learning 2000.*
- [27] ESKIN, E., ARNOLD, A., PRERAU, M., PORTNOY, L., and STOLFO, S., 2002. A geometric framework for unsupervised anomaly detection. In *Applications of Data Mining in Computer Security* Springer, 77-101.
- [28] FAN, W., MILLER, M., STOLFO, S., LEE, W., and CHAN, P., 2004. Using artificial anomalies to detect unknown and known network intrusions. *Knowledge and Information Systems* 6, 5, 507-527.
- [29] FERRANTE, J., OTTENSTEIN, K.J., and WARREN, J.D., 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3, 319-349.
- [30] FU, X. and QIAN, K., 2008. SAFELI: SQL Injection Scanner Using Symbolic Execution. *In Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, 34-39.
- [31] GARDIKIOTIS, S., LAZAROU, V., and MALEVRIS, N., 2007. An Agent-based Approach for the Maintenance of Database Applications. *In Proceedings of International Conference on Software Engineering Research, Management & Applications, 2007*, IEEE, 558-568.
- [32] GARDIKIOTIS, S.K. and MALEVRIS, N., 2009. A two-folded impact analysis of schema changes on database applications. *International Journal of Automation and Computing* 6, 2, 109-123.
- [33] GHOTING, A., PARTHASARATHY, S., and OTEY, M.E., 2008. Fast mining of distance-based outliers in high-dimensional datasets. *Data Mining and Knowledge Discovery* 16, 3, 349-364.
- [34] GOULD, C., SU, Z., and DEVANBU, P., 2004. Static checking of dynamically generated queries in database applications. *In Proceedings of 26th International Conference on Software Engineering*, 2004, 645-654.
- [35] HAN, J., PEI, J., and YIN, Y., 2000. Mining frequent patterns without candidate generation. *In ACM SIGMOD Record*, 1-12.
- [36] HENRARD, J. and HAINAUT, J.-L., 2001. Data dependency elicitation in database reverse engineering. *In Proceedings of Fifth European Conference on Software Maintenance and Reengineering*, 2001. IEEE, 11-19.
- [37] HUANG, Y.W., YU, F., HANG, C., TSAI, C.H., LEE, D.T., and KUO, S.Y., 2004. Securing Web Application Code by Static Analysis and Runtime Protection. *In Proceedings of the 13th International Conference on World Wide Web*, 40-52.

- [38] JIAWEI, H. and KAMBER, M., 2001. Data mining: concepts and techniques. San Francisco, CA, itd: Morgan Kaufmann 5.
- [39] JONES, J.A., HARROLD, M.J., and STASKO, J., 2002. Visualization of test information to assist fault localization. *In Proceedings of the 24th International Conference on Software Engineering* ACM, 467-477.
- [40] JOVANOVIĆ, N., KRUEGEL, C., and KIRDA, E., 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. *In Proceedings of IEEE Symposium on Security and Privacy*, 2006, 6 pp.-263.
- [41] JOVANOVIĆ, N., KRUEGEL, C., and KIRDA, E., 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. *In Proceedings of 2006 IEEE Symposium on Security and Privacy IEEE Computer Society*.
- [42] JOVANOVIĆ, N., KRUEGEL, C., and KIRDA, E., 2006. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. *In Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, 27-36.
- [43] KAPFHAMMER, G.M. and SOFFA, M.L., 2003. A family of test adequacy criteria for database-driven applications. *ACM SIGSOFT Software Engineering Notes* 28, 5, 98-107.
- [44] KHAN, L., AWAD, M., and THURASINGHAM, B., 2007. A new intrusion detection system using support vector machines and hierarchical clustering. *The International Journal on Very Large Data Bases* 16, 4, 507-521.
- [45] KIDWELL, B., 2011. A Decision Support System for The Classification of Software Coding Faults: A Research Abstract. *In Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, 1158-1160.
- [46] KOREL, B., 1987. The program dependence graph in static program testing. *Information Processing Letters* 24, 2, 103-108.
- [47] KROENKE, D. and AUER, D.J., 2010. Database concepts. Prentice Hall.
- [48] LAW, J. and ROTHERMEL, G., 2003. Whole program path-based dynamic impact analysis. *In Proceedings of the 25th IEEE International Conference on Software Engineering*, 2003, 308-318.
- [49] LAZAREVIC, A., KUMAR, V., and SRIVASTAVA, J., 2005. Intrusion detection: A survey. *In Managing Cyber Threats Springer*, 19-78.
- [50] LEE, Y.W., PIPINO, L., STRONG, D.M., and WANG, R.Y., 2004. Process-embedded data integrity. *Journal of Database Management (JDM)* 15, 1, 87-103.
- [51] LENGAUER, T. and TARJAN, R.E., 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1, 121-141.

- [52] LI, Z. and ZHOU, Y., 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. *In ACM SIGSOFT Software Engineering Notes*, 306-315.
- [53] LIAO, C., QUINLAN, D., VUDUC, R., and PANAS, T., 2010. Effective source-to-source outlining to support whole program empirical optimization. *Languages and Compilers for Parallel Computing*, 308-322.
- [54] LIU, K. and TAN, H.B.K., 2012. Automated Insertion of Exception Handling for Key and Referential Constraints. *In Proceedings of Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, 848-856.
- [55] LIU, K., TAN, H.B.K., and CHEN, X., 2011. Extraction of Attribute Dependency Graph from Database Applications. *In Proceedings of Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, 138-145.
- [56] LIU, K., TAN, H.B.K., and CHEN, X., 2013. Aiding Maintenance of Database Applications Through Extracting Attribute Dependency Graph. *Journal of Database Management (JDM)* 24, 1, 20-35.
- [57] LIU, K., TAN, H.B.K., and CHEN, X., 2013. Automated Insertion of Exception Handling for Key and Referential Constraints. *Journal of Database Management (JDM)* 24, 1, 1-19.
- [58] LIU, K., TAN, H.B.K., CHEN, X., ZHANG, H., and PADMANABHUNI, B., 2011. Automated Extraction of Data Lifecycle Support from Database Applications. *In Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011)*, July 7-9, 432-437.
- [59] LIU, K., TAN, H.B.K., and LEANIDAVICH, A.Y. Detecting Anomaly in Database Attributes Maintenance and Usage. *Proceedings of the 26rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2014)*, July 7-9, 705-710.
- [60] LIVSHITS, B. and ZIMMERMANN, T., 2005. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. *In ACM SIGSOFT Software Engineering Notes*, 296-305.
- [61] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V.J., and HAZELWOOD, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. *In ACM Sigplan Notices*, 190-200.
- [62] MADIRAJU, P., SUNDERRAMAN, R., NAVATHE, S.B., and WANG, H., 2006. Semantic integrity constraint checking for multiple XML databases. *Journal of Database Management (JDM)* 17, 4, 1-19.

- [63] MARTEL, M., 2009. Program transformation for numerical precision. *In Proceedings of the 2009 ACM SIGPLAN Workshop On Partial Evaluation And Program Manipulation*, 101-110.
- [64] MAULE, A., EMMERICH, W., and ROSENBLUM, D., 2008. Impact analysis of database schema changes. *In Proceedings of ACM/IEEE 30th International Conference on Software Engineering*, 2008, 451-460.
- [65] MENS, T. and TOURW, T., 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30, 2, 126-139.
- [66] MENZIES, T., GREENWALD, J., and FRANK, A., 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1, 2-13.
- [67] NAGY, C., P NTOS, J., GERGELY, T., and BESZ DES, Á., 2010. Towards a Safe Method for Computing Dependencies in Database-Intensive Systems. *In Proceedings of 14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010 IEEE, 166-175.
- [68] NETHERCOTE, N. and SEWARD, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *In ACM Sigplan notices*, 89-100.
- [69] OLDMEADOW, J., RAVINUTALA, S., and LECKIE, C., 2004. Adaptive clustering for network intrusion detection. *In Advances in Knowledge Discovery and Data Mining* Springer, 255-259.
- [70] ORDONEZ, C. and GARC A-GARC A, J., 2008. Referential Integrity Quality Metrics. *Decision Support Systems* 44, 2, 495-508.
- [71] ORSO, A., SINHA, S., and HARROLD, M.J., 2004. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 13, 2, 199-239.
- [72] OWASP, X., 2009. prevention Cheat-sheet.
- [73] PARR, T. and FISHER, K., 2011. LL (*): The foundation of the ANTLR parser generator. *In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 425-436.
- [74] PARR, T. and OTHERS, 2009. ANTLR parser generator. Online Stand Dezember.
- [75] PHC, 2012. The Open Source PHP Compiler.
- [76] PODGURSKI, A., LEON, D., FRANCIS, P., MASRI, W., MINCH, M., SUN, J., and WANG, B., 2003. Automated support for classifying software failure reports. *In Proceedings of 25th International Conference on Software Engineering*, 2003, 465-475.

- [77] PORTNOY, L., ESKIN, E., and STOLFO, S., 2001. Intrusion detection with unlabeled data using clustering. *In Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA)*, 2001 Citeseer.
- [78] RENIERES, M. and REISS, S.P., 2003. Fault localization with nearest neighbor queries. *In Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 2003, 30-39.
- [79] ROBERTSON, W.K., MAGGI, F., KRUEGEL, C., and VIGNA, G., Effective Anomaly Detection with Scarce Training Data. *In NDSS*, 2010.
- [80] RSnake, SQL injection cheat sheet Esp: for filter evasion.
- [81] RSnake, 2014. XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion.
- [82] SCH LKOPF, B., PLATT, J.C., SHAW-TAYLOR, J., SMOLA, A.J., and WILLIAMSON, R.C., 2001. Estimating the support of a high-dimensional distribution. *Neural Computation* 13, 7, 1443-1471.
- [83] SCHULTZ, M.G., ESKIN, E., ZADOK, E., and STOLFO, S.J., 2001. Data mining methods for detection of new malicious executables. *In Proceedings of IEEE Symposium on Security and Privacy*, 2001, 38-49.
- [84] SHOHAM, S., YAHAV, E., FINK, S.J., and PISTOIA, M., 2008. Static Specification Mining Using Automata-based Abstractions. *IEEE Transactions on Software Engineering* 34, 5, 651-666.
- [85] SILBERSCHATZ, A., KORTH, H.F., and SUDARSHAN, S., 2002. Database System Concepts. McGraw-Hill.
- [86] SMITH, R., BIVENS, A., EMBRECHTS, M., PALAGIRI, C., and SZYMANSKI, B., 2002. Clustering approaches for anomaly based intrusion detection. *In Proceedings of Intelligent Engineering Systems Through Artificial Neural Networks*, 579-584.
- [87] STEINWART, I., HUSH, D.R., and SCOVEL, C., 2005. A classification framework for anomaly detection. *In Journal of Machine Learning Research*, 211-232.
- [88] STROULIA, E. and SYST, T., 2002. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review* 10, 1, 8-17.
- [89] SU REZ-CABAL, M.J. and TUYA, J., 2004. Using an SQL coverage measurement for testing database applications. *In ACM SIGSOFT Software Engineering Notes*, 253-262.
- [90] TAN, H.B.K., LING, T.W., and GOH, C.H., 2002. Exploring into programs for the recovery of data dependencies designed. *IEEE Transactions on Knowledge and Data Engineering*, 14, 4, 825-835.
- [91] TAN, H.B.K. and THEIN, N.L., 2004. Recovery of PTUIE handling from source codes through recognizing its probable properties. *IEEE Transactions on Knowledge and Data Engineering*, 16, 10, 1217-1231.

- [92] TAN, H.B.K., ZHAO, Y., and ZHANG, H., 2009. Conceptual data model-based software size estimation for information systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 19, 2, 4.
- [93] THOMAS, S., WILLIAMS, L., and XIE, T., 2009. On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities. *Information and Software Technology* 51, 3, 589-598.
- [94] TUYA, J., SU\A, R.-C., MA JOS\E., and LA RIVA, C.D., 2007. Mutating database queries. *Information and Software Technology* 49, 4, 398-417.
- [95] TUYA, J., SUAREZ-CABAL, M.J., and DE LA RIVA, C., 2006. SQLMutation: A tool to generate mutants of SQL database queries. *In Proceedings of the Second Workshop on Mutation Analysis*, 2006, 1-1.
- [96] VAN DEN BRINK, H., VAN DER LEEK, R., and VISSER, J., 2007. Quality assessment for embedded sql. *In Proceedings of Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, 163-170.
- [97] VAN DEURSEN, A. and KUIPERS, T., 1998. Rapid system understanding: Two COBOL case studies. *In Proceedings of the 6th International Workshop on Program Comprehension*, IEEE, 90-97.
- [98] VIGNA, G., VALEUR, F., BALZAROTTI, D., ROBERTSON, W., KRUEGEL, C., and KIRDA, E., 2009. Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and SQL queries. *Journal of Computer Security* 17, 3, 305-329.
- [99] WASSERMANN, G., YU, D., CHANDER, A., DHURJATI, D., INAMURA, H., and SU, Z., 2008. Dynamic test input generation for web applications. *In Proceedings of the 2008 International Symposium on Software Testing And Analysis* ACM, 249-260.
- [100] WATSON, A.H., MCCABE, T.J., and WALLACE, D.R., 1996. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication* 500, 235, 1-114.
- [101] WIEDERMANN, B., IBRAHIM, A., and COOK, W.R., 2008. Interprocedural query extraction for transparent persistence. *In ACM Sigplan Notices*, 19-36.
- [102] WIKIPEDIA, 2012. Extended Backus-CNaur Form.
- [103] WIKIPEDIA, 2015. Data mining.
- [104] WIKIPEDIA, 2015. Static program analysis.
- [105] WILLMOR, D. and EMBURY, S.M., 2006. An intensional approach to the specification of test cases for database applications. *In Proceedings of the 28th International Conference on Software Engineering* ACM, 102-111.

- [106] XIE, Y. and YU, S.-Z., 2009. A large-scale hidden semi-Markov model for anomaly detection on user browsing behaviors. *IEEE/ACM Transactions on Networking*, 17, 1, 54-65.
- [107] YU, H., HAN, J., and CHANG, K.-C., 2004. PEBL: Web page classification without negative examples. *IEEE Transactions on Knowledge and Data Engineering*, 16, 1, 70-81.
- [108] ZHANG, H., TAN, H.B.K., ZHANG, L., LIN, X., WANG, X., ZHANG, C., and MEI, H., 2011. Checking Enforcement of Integrity Constraints in Database Applications Based on Code Patterns. *Journal of Systems and Software*.
- [109] ZHANG, J. and WANG, H., 2006. Detecting outlying subspaces for high-dimensional data: the new task, algorithms, and performance. *Knowledge and Information Systems* 10, 3, 333-355.
- [110] ZHANG, J., XU, C., and CHEUNG, S., 2001. Automatic generation of database instances for white-box testing. *In Proceedings of 25th Annual International IEEE Computer Software and Applications Conference*, 2001, 161-165.
- [111] ZHOU, C. and FRANKL, P., 2009. Mutation testing for Java database applications. *In Proceedings of International Conference on Software Testing Verification and Validation*, 2009, 396-405.