University of Namur

THESIS / THÈSE

DOCTOR OF SCIENCES

Understanding Data-Intensive Systems Through The Analysis of SQL Execution Traces

Noughi, Nesrine

Award date:
2018

Awarding institution:
University of Namur

Link to publication

University of Namur

PReCISE Research Center

Doctor of Philosophy
Discipline : Computer Science

presented by

Nesrine Noughi

Understanding Data-Intensive Systems Through The Analysis of SQL Execution Traces

Prof. Anthony Cleve	University of Namur	Promotor
Prof. Naji Habra	University of Namur	Co-promotor
Prof. Wim Vanhoof	University of Namur	President
Prof. Vincent Englebert	University of Namur	Internal Reviewer
Prof. Tom Mens	University of Mons	External Reviewer
Dr. Alexander Serebrenik	Eindhoven University of Technology	External Reviewer

Abstract

Most software systems need to be adapted during their life-cycle. It is estimated that more than 60% of the cost of a software system is related to its maintenance and evolution. The evolution of a software system not only involves the modification made to its various components, but it also includes an indispensable preliminary phase, which is the in-depth understanding of each component of the system. This is especially true for a certain type of systems called data-intensive systems. Within such a system, the interactions between the application programs and the database are becoming increasingly difficult to analyze, and therefore to understand.

This is why program understanding in general has become a important topic of interest for software engineering researchers and developers. In particular, understanding today's data-intensive systems clearly calls for automated support.

The goal of this thesis is to facilitate the understanding of large data-intensive systems using dynamic analysis, visualization and process mining techniques. The dynamic analysis techniques seek to analyze and visualize the data-manipulation behavior of data-intensive systems via the analysis of their SQL execution traces. The process mining techniques seek to retrieve recurring patterns and extract the data-manipulation processes followed by the program itself.

Résumé

La plupart des systèmes logiciels doivent être adaptés au cours de leur vie. On estime que plus de 60% du coût d'un logiciel est lié à sa maintenance et à son évolution. L'évolution d'un logiciel ne consiste pas seulement en la modification apportées à ses différents composants, mais elle comprend aussi une phase préliminaire indispensable qui est la compréhension en profondeur de chaque composant du système. C'est d'autant plus vrai pour une certaine catégorie de systèmes, tel que les systèmes à forte intensité de données (Data-intensive systems). Dans ce genre de systèmes, l'interaction entre les programmes d'application et la base de données est de plus en plus difficile à analyser, et donc à comprendre.

De ce fait, la compréhension de programmes est devenue aujourd'hui une réelle préoccupation des chercheurs et développeurs. Car il s'agit d'un problème complexe, en particulier dans le cas de ce type de systèmes qui nécessitent l'utilisation de méthodes et techniques automatisées et ingérées aux sein des environnements de développement de logiciels.

Cette thèse a pour objectif l'amélioration de la compréhension de ces grands systèmes à l'aide des techniques de l'analyse dynamique, de la visualisation de programmes et de l'exploration de processus. La technique d'analyse dynamique a pour but d'analyser et de visualiser le comportement de la manipulation de données d'un programme d'application via l'analyse de leurs traces d'exécution SQL. La technique d'exploration de processus a pour objectif de retrouver des patterns récurrents et d'extraire le workfow de la manipulation de données suivi par le programme.

Acknowledgments

First of all, I would like to thank God for making all things possible for me and giving me strengthen to carry out this research work.

I would like to express my sincere gratitude to my advisor Prof. Anthony Cleve for the continuous support during this thesis, his valuable advice, and enthusiasm. His guidance helped me to acquire new understanding and extend my experiences. He was not only an advisor but also a good listener and a friend who showed me what I was able to achieve even when I did not see it myself.

Besides my advisor, I would like to thank all the members of the jury, for their insightful comments and encouragement, but also for the hard questions which prompted me to widen my research from various perspectives. Namely, I would like to thank Prof. Wim Vanhoof, the president of the jury, Prof. Naji Habra, my co-promotor and internal member of the jury, Prof. Vincent Englebert, internal member of the jury, and Prof. Tom Mens and Prof. Alexander Serebrenik external members of the jury.

Special thanks to my colleague Csaba Nagy for his help and encouragements to give me back my confidence. Anne-France Brogneaux, Jonathan Lemaitre, Marco Mori, Javier Bermúdez, Loup Meurice, Adrien Bibal, Maxime Gobert, Minh Vu my officemates for the stimulating discussions, and for all the fun we have had in the last seven years. Also, I thank all my colleagues for the all good times spent together in every coffee break which made the Computer Science Faculty a great place to work, as well as all my friends (Fadhela, Hanane, Irina, Hajer, Nesrine, Abdelmounaim, George) who have given me their moral support during this year of study, I thank them sincerely

Last but not the least, the simplest words being the strongest, I address all my affection to my family, and especially to my parents. Despite my distance since (too) many years, their confidence, their tenderness, their love carries me and guide me every day. Thank you for making me what I am today. I love you.

Finally, a big thank you to the people who believed in me and who allowed me to reach the end of this thesis.

à la mémoire de mon cher papa
Essaid Noughi

à ma chère maman,
à ma chère famille

Contents

Abstract i

Résumé iii

1 Introduction 1
- 1.1 Research Context 1
- 1.2 Objective and Research Questions 2
- 1.3 Contributions 3
- 1.4 Thesis Outline 4
- 1.5 Publications 7

I Research Domain and Related Literature 9

2 Research Domain 11
- 2.1 Introduction 11
- 2.2 Software Maintenance and Evolution 12
- 2.3 Program Comprehension 14
- 2.4 Data-Intensive Systems 15
- 2.5 Database Engineering 16
- 2.6 Database Reverse Engineering 18
- 2.7 The Generic Entity-Relationship Model 20
  - 2.7.1 Conceptual Schema 20
  - 2.7.2 Logical Schema 21
- 2.8 Program Analysis 22
  - 2.8.1 Static Program Analysis 23
  - 2.8.2 Dynamic Program Analysis 25
  - 2.8.3 Static Analysis vs. Dynamic Analysis 27
- 2.9 Software Visualization 27
  - 2.9.1 Definition 28
  - 2.9.2 Classification of Software Visualization 28

2.9.3 How Do We Choose The Right Visualization? 29
2.9.4 2D vs. 3D Visualization 30
2.10 Process Mining 31
2.11 Conclusions 32

3 Related Literature 35
3.1 Introduction 35
3.2 Program Analysis for Program Comprehension 36
3.2.1 Dynamic Analysis for Program Comprehension 36
3.2.2 SQL Statement Analysis 39
3.3 Process Mining for Program Comprehension 44
3.3.1 Synthesis 45
3.4 Visualization for Program Comprehension 46
3.4.1 Synthesis 49
3.5 Conclusions 49

II Dynamic Analysis of SQL Execution Traces for Data-Intensive Systems 51

4 Overview of The Framework 53
4.1 Introduction 53
4.2 Motivation 54
4.3 Preliminaries 55
4.3.1 Mapping between Conceptual and Logical Schemas . . 55
4.3.2 SQL Execution Traces 56
4.4 Problem Statement 57
4.5 Approach Definition 57
4.5.1 Phase 1: Intra-scenario Analysis 58
4.5.2 Phase 2: Inter-scenario Analysis 59
4.5.3 Phase 3: Code Re-documentation 60
4.6 Conclusions 60

5 Understanding the Data Manipulation Behavior from SQL Execution Traces 61
5.1 Introduction 61
5.2 Approach and Research Questions 62
5.3 Trace Capturing 63
5.3.1 Query Interception 64
5.3.2 Query Parsing 67
5.4 Intra-Query Analysis 69

5.4.1 Sub-schema Extraction: 69
5.4.2 Object Frequency 70
5.4.3 Synthesis 71
5.5 Inter-Query Analysis 71
5.5.1 Dependency Extraction 72
5.5.2 Loop Detection 74
5.5.3 Synthesis 77
5.6 Query Interpretation 78
5.6.1 Trace Abstraction 78
5.6.2 Subschema Annotation 80
5.6.3 Interpretation Generation 81
5.7 Tool Support 82
5.7.1 Metrics 84
5.7.2 DB-MAIN 84
5.7.3 JUNG Library 84
5.7.4 DAViS User Interface 85
5.7.5 Visualization Modes 87
5.7.6 Component 1: Intra-query Analysis Visualization . . . 89
5.7.7 Component 2: Inter-query Analysis Visualization . . . 90
5.7.8 Component 3: Query Interpretation Visualization . . . 91
5.8 Case Studies 93
5.8.1 Traces Capturing 95
5.8.2 Queries Parsing 96
5.8.3 Sub-schemas Results 97
5.8.4 Dependency Results 101
5.9 Conclusions 102

6 Extracting Data Manipulation Processes from SQL Execution Traces 105
6.1 Introduction 105
6.2 Illustrative Scenario 106
6.3 Approach and Research Questions 108
6.4 Data-oriented Properties 110
6.4.1 Query Filtering 110
6.4.2 Associating Properties 111
6.4.3 Query Clustering 112
6.4.4 Cluster Labeling 113
6.5 Process Mining 117
6.5.1 Traces Abstraction 117
6.5.2 Process Extraction 118
6.6 Evaluation 121

6.6.1 Scenario Used 122
6.6.2 Experiments Description 123
6.6.3 Threats To Validity 126
6.7 Conclusions 127

7 Code Re-documentation from SQL Execution Traces 129
7.1 Introduction 129
7.2 Motivation 130
7.3 Approach and Research Questions 131
7.4 Preliminaries 132
7.4.1 Java DataBase Connectivity (JDBC) 132
7.4.2 Extraction Techniques of SQL Trace Locations 133
7.4.3 JAVA Abstract Syntax Tree (AST) 134
7.5 Comments Generation 135
7.5.1 Information Determination 135
7.5.2 Information Gathering 138
7.6 Comments Injection 138
7.7 Discussions 139
7.8 Illustrative Example 141
7.9 Results 143
7.10 Conclusions 144

III Evaluation and Validation 147

8 An Empirical Study on the Use of SQL Execution Traces for Program Comprehension 149
8.1 Introduction 149
8.2 Related Work 150
8.3 Experimental Description 151
8.3.1 Initial Considerations 151
8.3.2 Evaluation Questions and Hypotheses 152
8.3.3 Used Scenario 152
8.3.4 Tasks Design 153
8.3.5 Pilot Studies 154
8.3.6 Experiment Procedure 155
8.3.7 Data Collection 156
8.4 Experiment Execution 157
8.4.1 Selection of Participants 157
8.4.2 Concrete Setting 157
8.5 Results 159

8.5.1 EQ1: Does DAViS reduce the time needed to complete the tasks? 160
8.5.2 EQ2: Does the educational background influence the usability of DAViS? 163
8.5.3 EQ3: Does DAViS increase the correctness of the tasks? 163
8.6 Discussion 164
8.6.1 Reasons for Different Time Requirements 165
8.6.2 Reasons for The Education Level Independence 168
8.6.3 Reasons for Correctness Differences 168
8.7 Threats to Validity 168
8.7.1 Internal Validity 169
8.7.2 External Validity 169
8.8 Conclusions 170

IV Conclusions 171

9 Conclusions 173
9.1 Summary of The Contributions 173
9.2 Future Challenges 179

List of Figures

1.1 General structure of the thesis. 5

2.1 The approximate cost of each phase of the software life-cycle [77] 13
2.2 Software maintenance types according to the authors of [42] . 14
2.3 An example of data-intensive systems architecture considered in this thesis 16
2.4 Database and application design processes 17
2.5 Database reverse engineering design processes 19
2.6 A conceptual schema example 20
2.7 The logical schema example and rough relational translation of the conceptual schema of Figure 2.6 22
2.8 Programs analysis techniques 23
2.9 Types of software visualization [47] 28
2.10 Types of process mining 31

4.1 General framework 54
4.2 An example of a mapping between a logical schema and a conceptual schema. 56
4.3 An example of a SQL execution trace + result of query q_1. . . 57
4.4 An overview of the framework 58

5.1 Bottom-up approach: Intra-scenario analysis. 62
5.2 The conceptual schema of a Customer-Order application . . . 65
5.3 The logical schema of a Customer-Order application 65
5.4 An example of an SQL trace extracted from Customer-Order application + result of query q_1 66
5.5 Sub-schema affected by the trace of Listing 5.4 71
5.6 The loop detection results of the SQL queries of trace 5.4. . . 77
5.7 Overall framework of DAViS 83
5.8 DAViS user interface 86

5.9 The global visualization of the trace of Figure 5.4 87
5.10 The local visualization of the first SQL statement of the SQL trace 5.4 (Left: the input, Right: the output visualization) . . 89
5.11 The sub-schema affected by the SQL execution trace of the trace of Figure 5.4 90
5.12 Loop visualization 91
5.13 Trace abstraction and visual interpretation 92
5.14 Natural language interpretation 93
5.15 Logical schema of WebCampus application 94
5.16 Logical schema of WebDeb application 95
5.17 Left: The logical schema of WebCampus, right: The subschema of the install_applet scenario 98
5.18 Left: The logical schema of WebDeb, right: The subschema of the validate affirmation scenario 99
5.19 The detected dependencies of the *install_applet* senario of WebCampus 99
5.20 Left: The foreign key dependency of the *Install_applet* scenario, right: The join query dependency of the *Install_applet* scenario 101
5.21 Left: The input-input dependency of the *Install_applet* scenario, right: The joint access dependency of the *Install_applet* scenario 101
5.22 The detected loops of the *install_applet* scenario of WebCampus102

6.1 The logical schema (Web-Store case study) 107
6.2 The conceptual schema (Web-Store case study) 107
6.3 Web Store: Three SQL execution traces 108
6.4 Bottom-up approach: Inter-scenario analysis. 109
6.5 Web Store : Abstracted Traces of SQL statements 118
6.6 Web Store: Process mined with *Trace 1*, *2* and *3*. 120
6.7 *RestaurantReservation*: Mined process (top) and process corrected by the designer (bottom). 124
6.8 e-Restaurant case study: Average recall measure (a) and average precision measure (b) of the mined process models depending on log coverage (1-trace logs, 2-trace logs, ..., 6-trace logs). 125

7.1 Bottom-up approach: A code re-documentation 131
7.2 The Abstract Syntax Tree workflow [38] 134
7.3 Conceptual schema of AcadYearManager application 141
7.4 The logical schema of an AcadYearManager application 142

7.5 A fragment of the collected SQL trace 142
7.6 The intra-scenario analysis results 143
7.7 The loop detection results 144
7.8 The abstraction of the first detected loop 144

8.1 Handouts for each group 156
8.2 Interaction diagram of round 1 for the between-subject factor tool and the within-subject factor task 162
8.3 Interaction diagram of round 2 for the between-subject factor tool and the within-subject factor task 162

9.1 How the contributions are connected 178

List of Tables

2.1 Characteristics of static and dynamic analysis for program analysis . . . 27
2.2 Characteristics of 2D and 3D visualization . . . 30

3.1 Summary of research works focusing on program comprehension 37
3.2 Summary of research works focusing on SQL statement analysis 43
3.3 Summary of research studies focusing on visualization . . . 48

5.1 Example of the parser structure . . . 68
5.2 The object frequency of the trace of Listing 5.4 . . . 72
5.3 The clusters of the SQL queries of trace 5.4 . . . 76
5.4 Abstraction rules of nested queries . . . 78
5.5 Visual metaphors used by DAViS . . . 85
5.6 The systems description . . . 94
5.7 Characteristics of the parsed queries of the 14 scenarios of WebCampus application . . . 96
5.8 Characteristics of the parsed queries of the 6 scenarios of WebDeb application . . . 97
5.9 The parser's results . . . 97
5.10 Statistics about the Webcampus subschemas analysis . . . 98
5.11 Statistics about the WebDeb subschemas analysis . . . 98
5.12 The access frequencies of the tables of the scenario install_applet . . . 100
5.13 The access frequencies of the tables of the scenario validate affirmation . . . 100

6.1 The rules of associating properties according to the parser clauses . . . 111
6.2 Web Store: Properties of the traces of SQL statements . . . 112
6.3 Web Store: Properties of the SQL statements . . . 113
6.4 Clusters of SQL queries of the Web Store application 6.3 . . . 114

6.5 The rules of labeling for the select query 115
6.6 Web Store: Data manipulation functions and I/O parameters . 116
6.7 Example traces 119

7.1 Dependency comment template 136

8.1 Characteristics of the four scenarios in the given system 152
8.2 Comprehension tasks 153
8.3 Visualization proposed by DAViS 154
8.4 Experiment layout 154
8.5 Distribution of students in the two groups 157
8.6 Measured response times (in seconds) 158
8.7 T-tests on individual tasks (round 1) 161
8.8 T-tests on individual tasks (round 2) 163
8.9 Left: Exact Fisher tests on individual tasks about correctness (round 1), right: Exact Fisher tests on individual tasks about correctness (round 2) 164
8.10 Debriefing questionnaire: perceived task difficulty with DAViS 165
8.11 Debriefing questionnaire: perceived help of DAViS 165

9.1 Thesis contributions 174

Chapter 1

Introduction

Satisfaction lies in the effort, not in the attainment,
full effort is full victory.
Mahatma Gandhi

Contents

1.1 Research Context 1

1.2 Objective and Research Questions 2

1.3 Contributions 3

1.4 Thesis Outline 4

1.5 Publications 7

1.1 Research Context

Nowadays, software systems are not simple computer applications that we use in personal computers or at work. Many in fact are highly sophisticated, large and complex software packages that play an important role in society and the economy. Now, software systems exist almost everywhere and they often constitute the heart of business-critical activities. As a consequence, software engineering, defined as an engineering discipline that focuses on all aspects of software system development [37], has remained one of the main centers of interest for researchers over the past four decades.

In the software development process, several models for the software lifecycle are defined, which can be grouped into six main phases according to the Waterfall Model introduced by Royce [72]: (1) *Requirements*, describes the customer's needs and all their constraints; (2)*Analysis*, consisting in transforming requirements into models, schemas and business rules; (3) *Design*, resulting in the software architecture; (4) *Programming*, consisting of translating specifications into a programming language and integration of software; (5) *Testing*, the systematic discovery and debugging of defects; (6) *Operations*, the installation, migration, support, and maintenance of complete systems.

However, we live in world that is constantly and rapidly changing. This is why software systems have to evolve in order to adapt to their ever-changing environment. Therefore, software maintenance and evolution, defined as ”the modification of a software product after delivery to correct faults, to improve performance or to adapt the product to a modified environment” [68], are known to be major factors in the cost of the software development process [77]. More specifically, David Lo [46] has estimated that up to 50% of the costs of software maintenance are related to software understanding, which is a typical initial task of the phase that is required before making any changes.

Software documentation has long been regarded as the most important source of information for software understanding. Unfortunately, nowadays documenting application programs has become so outdated, economically unsustainable and psychologically unbearable. As a consequence, most of the existing systems are virtually undocumented or have an outdated documentation. Thus, maintaining and evolving such systems can be performed only if the system has been sufficiently understood in terms of its structure and behavior. To cope with this, understanding the objectives, the behavior and the internal artifacts of an existing and undocumented software system must be achieved in some another way. It therefore seems obvious that we need effective techniques and tools to support this task. To this end, numerous methods have been proposed that seek to analyze and understand a software system from its artifacts such as source code, architectural diagrams, design information, execution traces and event log.

1.2 Objective and Research Questions

However, as Cleve et al. [11] highlighted in a study on data-intensive systems challenges, very few of the recently proposed studies were interested in modern data-intensive systems that consist of a set of applications performing complex, continuous and dynamic interactions among the application programs and a large set of data, typically stored in a database. In such systems, the communication and relation with the database system are usually realized by the Structured Query Language (SQL), where the SQL queries are built dynamically at runtime, even through String concatenations or through an external view of data, such the very popular object-relational mapping (ORM) technique, which lets programmers apply an object-oriented view of the database. This programming technique is widely used in the Java world with various frameworks such as Hibernate, JPA and JGrinder. Therefore, the understanding of the database access behavior of a program, which has become an important (yet largely ignored) aspect of program comprehension constitutes an important part of the program behavior, which are important to consider and understand.

In the light of this, the main objective of this research line, which is the process of data-intensive systems understanding, is to define new framework to analyze the program-database interractions to help developers to understand data-intensive systems. To achieve this goal, we intend to rely on the intensive use of dynamic program analysis, with the help of visualization techniques and process mining to present the results of our cross-analysis techniques to the user. More specifically, we wish to recover and understand the data-manipulation behavior of data-intensive applications, by analyzing the information that can be captured from SQL execution traces.

To structure our methodology towards this objective, we need to answer the following questions that drive our research methodology:

RQ1: Can we automatically relate the program execution traces, program source code and the database schema to each other?

RQ2: Does an analysis of the data manipulation behavior support the understanding of data-intensive programs?

RQ3: How can we automatically extract a model of the data manipulation behavior?

RQ4: Can we automatically re-document this behavior in the program source code?

1.3 Contributions

During this study, we propose to examine the problematic areas associated with the understanding of data-intensive systems through the analysis of their data-manipulation behavior. This purpose is achieved through a list of contributions that seek to address the four research questions stated above.

1. **An automated approach for understanding data-manipulation behavior through the analysis of a single SQL execution trace:**

   - A parser that extracts and represents the most relevant information of an SQL execution trace in a unified data-object.
   - Algorithms that extract implicit dependencies between consecutive SQL statements.
   - A visual tool supported by the approach, implementing the various given algorithms.

2. **An automated approach for extracting a model of data manipulation behavior through the analysis of multiple SQL execution traces:**
   - Algorithms that extract, label and clusters data-manipulation functionalities to recover the followed data manipulation model.
3. **An automated approach for re-documenting the program source code through comments:**
   - An algorithm that generates textual comments to be injected into programs source code.
   - An algorithm that injects the generated comments.
4. **A two steps evaluation approach:**
   - Case studies conducted to demonstrate the practical usefulness and validate the correctness of the different proposed algorithms.
   - An empirical study on the use of SQL traces analysis to demonstrate the feasibility, efficiency and accuracy of some parts of our proposed approach.

1.4 Thesis Outline

The thesis contains nine chapters (including the Introduction and Conclusions chapters) organized in three main parts with an additional introduction and conclusions, as depicted in Figure 1.1. The remainder of this manuscript is organized as follows:

1. **Part 1: Research Domain and Related Literature**
   - **Chapter 2:** We present a background on our research context. we start by defining concepts related to software maintenance and evolution of data-intensive applications. Then, we summarize the most widely used methods and techniques for supporting program comprehension.

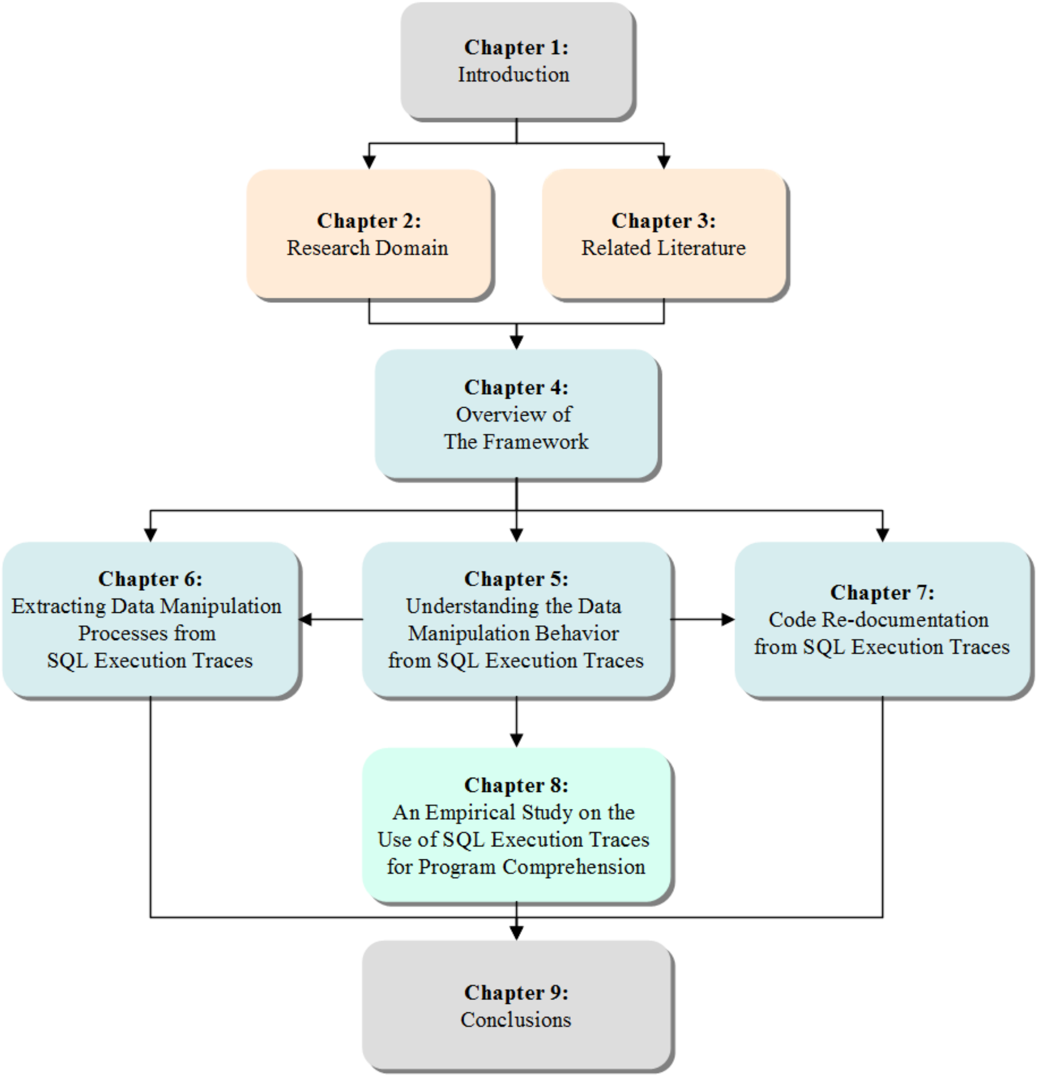


Figure 1.1: General structure of the thesis.

- **Chapter 3:** We summarize and discuss the different existing approaches and tools that assist software understanding and it is grouped into three categories according to the techniques used. It also places our work in relation to the most relevant approaches used.

2. **Part 2: Dynamic Analysis of SQL Execution Traces for Data-intensive Systems**

    - **Chapter 4:** We introduce a set of suitable definitions and preliminaries. Based on these concepts and definitions, we formally define our problem statement, research questions and approach in three parts.
    - **Chapter 5:** We present a new approach for understanding the data-manipulation behavior of programs based on an analysis of its data accesses via SQL queries (CRUD operations). It starts by formally defining the solution needed to achieve this goal in terms of successive levels of understanding and illustrates each level using a running example scenario. Then, it presents the tool supported by the proposed approach before concluding with two case studies.
    - **Chapter 6:** We propose a new approach for extracting data-manipulation processes through the analysis of multiple SQL execution traces of the same program scenario. The approach consists of a set of algorithms for extracting, clustering, labeling and merging data manipulation functions followed by the given scenario.
    - **Chapter 7:** In the last chapter of this part, we provide a new approach that shows how we can re-document the program source code with its data manipulation behavior through comments.

3. **Part 3: Evaluation and Validation**

    - **Chapter 8:** We describe an empirical study on the use of SQL execution traces. It starts by defining a general objective of the empirical study. Then, it defines the addressed hypothesis before presenting details of the experiment performed. Finally, we provide some evaluation results before drawing some conclusions.

4. **Part 4: Conclusions and Future Work**

    - **Chapter 9 :** Here, we seek to outline the general conclusions of this thesis. We start by summarizing the contributions of the thesis related to the above research questions. Then, we make some suggestions for future research needed to achieve our long-term objectives.

1.5 Publications

Most of the contributions presented in this thesis were published as peer-reviewed publications:

1. Nesrine Noughi, Stefan Hanenberg and Anthony Cleve. An Empirical Study on the Use of SQL Trace Visualization for Program Understanding. In IEEE International Conference on Software Quality, Reliability and Security Companion, (QRS-C 2017).

2. Nesrine Noughi, Stefan Hanenberg and Anthony Cleve. An Empirical Study on the Use of SQL Trace Visualization for Program Understanding. In Seminar series on Advanced Techniques & Tools for Software Evolution (SATToSE'16).

3. Nesrine Noughi and Anthony Cleve: Conceptual Interpretation of SQL Execution Traces for Program Comprehension. Proceedings of the 6th International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2015) - Colocated with the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015). IEEE Computer Society Pres.

4. Marco Mori, Nesrine Noughi and Anthony Cleve: Mining SQL Execution Traces for Data Manipulation Behavior Recovery. In CAiSE Forum 2014, Selected Extended Papers. (Lecture Notes in Business Information Processing).

5. Nesrine Noughi, Marco Mori, Loup Meurice and Anthony Cleve: Understanding the Database Manipulation Behavior of Programs - Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014): Early Research Achievement Track. IEEE Computer society.

6. Marco Mori, Nesrine Noughi and Anthony Cleve: Mining SQL Execution Traces for Data Manipulation Behavior Recovery. Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014): CAiSE forum track. CEUR-WS.org, Vol. 1164, p. 41-48 8 p.

7. Anthony Cleve, Nesrine Noughi and Jean-Luc Hainaut: Dynamic program analysis for database reverse engineering. In Ralf Lämmel, Joaõ Saraiva, and Joost Visser, editors, Generative and Transformational Techniques in Software Engineering, Volume 7680 of Lecture Notes in Computer Science, pages 297–321. Springer, 2013.

8. Nesrine Noughi, Anthony Cleve: Understanding Data Intensive Systems Using Dynamic Analysis and Visualization - International Conference on Software Maintenance (DS-ICSM 2013): Doctoral Symposium. IEEE Computer society.

9. Loup Meurice, Nesrine Noughi and Anthony Cleve. Visualizing SQL execution traces for program understanding. In Seminar series on Advanced Techniques & Tools for Software Evolution (SATToSE'13).

Part I

Research Domain and Related Literature

Chapter 2

Research Domain

The secret of getting ahead is getting started.
Mark Twain

Contents

2.1 Introduction 11
2.2 Software Maintenance and Evolution 12
2.3 Program Comprehension 14
2.4 Data-Intensive Systems 15
2.5 Database Engineering 16
2.6 Database Reverse Engineering 18
2.7 The Generic Entity-Relationship Model 20
2.8 Program Analysis 22
2.9 Software Visualization 27
2.10 Process Mining 31
2.11 Conclusions 32

2.1 Introduction

With software maintenance and evolution, the key ingredient is program comprehension. Program comprehension has become a primary task of the software maintenance and evolution phase mostly due to the absence of (sufficient, up-to-date) documentation. This is especially true in the case of data-intensive systems, which are characterized by the complex and extremely dynamic nature of interactions between the application programs and their underlying database.

In the light of this, numerous approaches have been proposed which use different techniques such as static analysis, dynamic analysis, process mining, and visualization, with the aim of improving the program comprehension phase of software systems.

In this chapter, we start by discussing program comprehension in software evolution and maintenance. Then, we present some fields that are relevant to program comprehension such as database engineering, database reverse engineering and the generic entity-relationship model, before defining data-intensive systems. Lastly, we present and discuss some techniques that we identified as being relevant to this problem such as program analysis, visualization and process mining.

2.2 Software Maintenance and Evolution

The software life-cycle refers to all the steps of software development from design to delivery. According to Schach et al. [77], most models of the software life-cycle include the following steps:

1. **Requirements engineering**, represents the expression, collection and formalization of the customer's needs and all their constraints. This step offers a formal specification of the general software architecture;

2. **Design**, consisting of defining precisely how the software will be constructed in order to meet the specifications agreed upon in the requirements specification document;

3. **Programming**, consisting of translating the functionalities of the design phase into a programming language. In order to handle the complexity of the programming process, the program is usually divided into separate units called modules;

4. **Integration**, consisting of ensuring that the combination of the different modules form the integrated software product;

5. **Delivery**, which is summarized by the delivery of the completed software product to the customer. Then, the customer will conduct acceptance testing in order to validate whether it meets the specifications agreed upon in the requirements specification document;

6. **Maintenance**, includes all corrective (corrective maintenance) such as bug fixes, correct real-world unidentified problems and scalable (scalable maintenance) actions on the software such as adding new functionality.

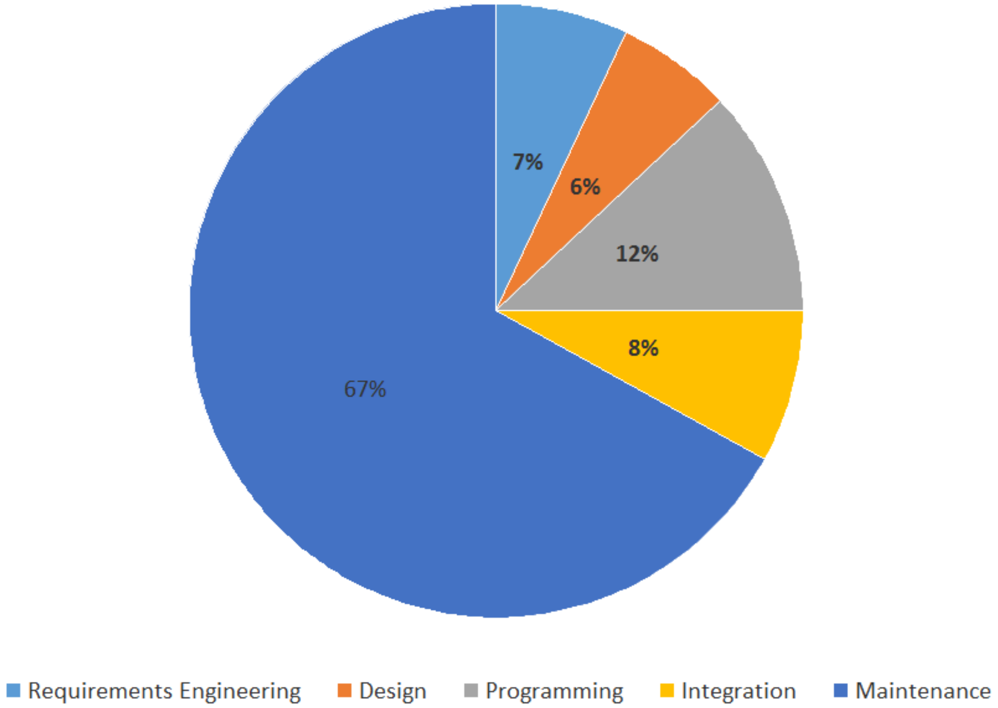


Figure 2.1: The approximate cost of each phase of the software life-cycle [77]

Schach et al. [77] have estimated the approximate relative costs of the phases of the software life-cycle. Figure 2.1 describes the relative cost of each task. From the figure, we can see that the maintenance step greatly dominates the cost of the software life-cycle. In the light of this, many studies have focused on this phase of the software life-cycle to define approaches that seek to reduce this cost and help developers to maintain systems.

Software maintenance and evolution processes often necessitate the recovery of a sufficient understanding of the software system, before the latter can be adapted to new or changing requirements. According to Lientz et al. [42], software maintenance and evolution includes four types, these being: (1) *corrective*, which concerns bugs-fixing; (2) *perfective*, concerns all changes in the software that occur when we add new functionalities; (3) *adaptive*, concerns all changes in the software that take place to enable it to adapt to the new

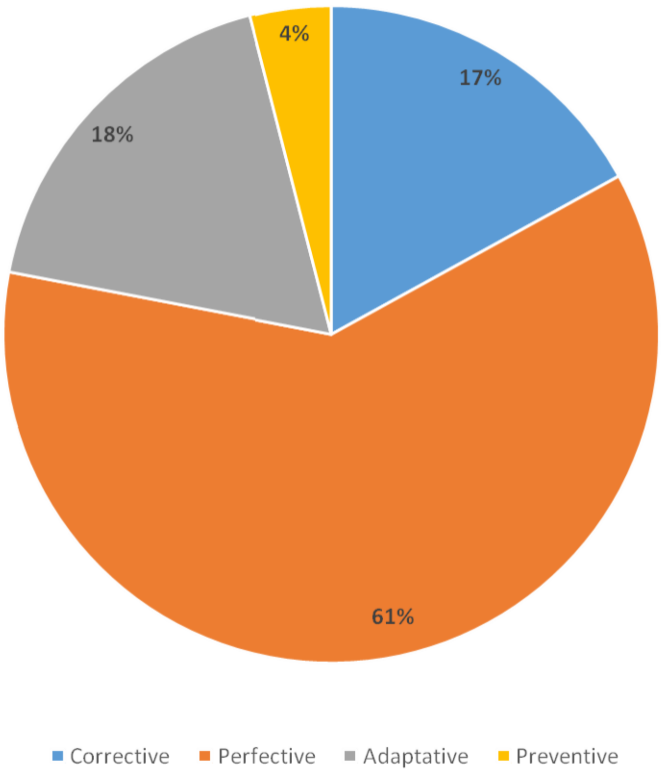


Figure 2.2: Software maintenance types according to the authors of [42]

environment - for instance, run the software on a new operating system and (4) *preventive*, this type consisting of implementing changes to prevent the occurrence of errors.

In this thesis, we also focus on the maintenance and evolution phase of the software life-cycle. More specifically, we will focus on its first activity, which is program comprehension.

2.3 Program Comprehension

Program understanding is the first task of each software maintenance type. Indeed, it is necessary to achieve a certain level of insight into the application before the application can be maintained. Corbi et al. [13] estimated that program understanding takes up to 50-60% of the maintenance time. Hence, improving the efficiency of this task may be regarded as a significant time saving in the entire maintenance phase.

With this in mind, several approaches have been proposed to support program comprehension, which can be grouped into two main categories, according to the following development model:

1. **Top-down approach**: A top-down approach is an overview of the system that is formulated by specifying, but without detailing, any first-level subsystem. Each subsystem is then refined in yet greater detail, sometimes with many additional subsystem levels, until the entire specification is reduced to base elements. With the program comprehension approach, the top-down understanding approach is typically chosen when developers are familiar with the system (in terms of code, problem domain and/or solution space). For instance, when source code is implemented by the developer who wants to maintain this system. Therefore, he already has absolute control over the code that had performed more or less the same functionalities. In this case, these similarities in code structure are easier to recognize in a top-down understanding process.

2. **Bottom-up approach**: In the bottom-up approach, the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which in turn are linked, sometimes in many levels, until a complete top-level system is formed. In the program comprehension model, the bottom-up understanding approach is sufficient when the developer who wants to change the system is completely unfamiliar with it.

Nevertheless, these approaches have some similarities with the analysis techniques that they use to support program understanding (e.g. dynamic analysis and static analysis), which we will present later.

2.4 Data-Intensive Systems

In this thesis, we are mainly interested in data-intensive systems. By data-intensive systems, we mean all systems that rely on intensive interactions between the application programs and their database. Figure 2.3 shows an example of the given data-intensive systems architecture. Such systems have two main characteristics:

1. They have to manage an increasing amount of data that are usually stored in databases;

2. Databases often occupy a central place;

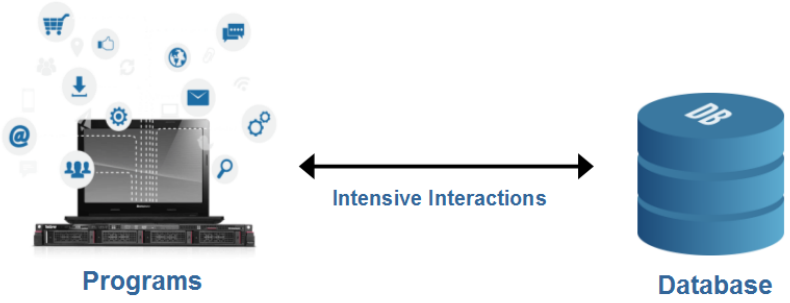


Figure 2.3: An example of data-intensive systems architecture considered in this thesis

3. The communication between the application programs and their database becomes increasingly important and dynamic;

Therefore, the evolution of data-intensive systems clearly requires new and better-adapted techniques to deal with new challenges of these systems, such as data capturing, data analysis, visualization, querying, updating and information privacy. This is especially true in the maintenance and the evolution phases, where the understanding task is treated as a complex problem. Because of this, *program comprehension of data-intensive systems* has now become a real research interest for researchers and developers.

2.5 Database Engineering

As mentioned in the last section, *databases* occupy a central place in data-intensive systems. Now, we are going to introduce the important concepts related to databases - more specifically, relational databases. A relational database is a set of tables containing a collection of related data grouped into predefined categories in order to make data access easier. They are managed by so-called database management systems (DBMS), more specifically, relational database management systems (RDBMS), which are software packages for creating, managing, manipulating and retrieving data from a relational database [30].

We focused on the relational model because historically the first database model that solved the issue of inconsistency, redundancy, concurrency and other problems was the relational model. In addition, the most widespread database management systems are relational [69].

Figure 2.4 depicts the main phases of database design connected with the application design phases. Here, we will just focus on the database

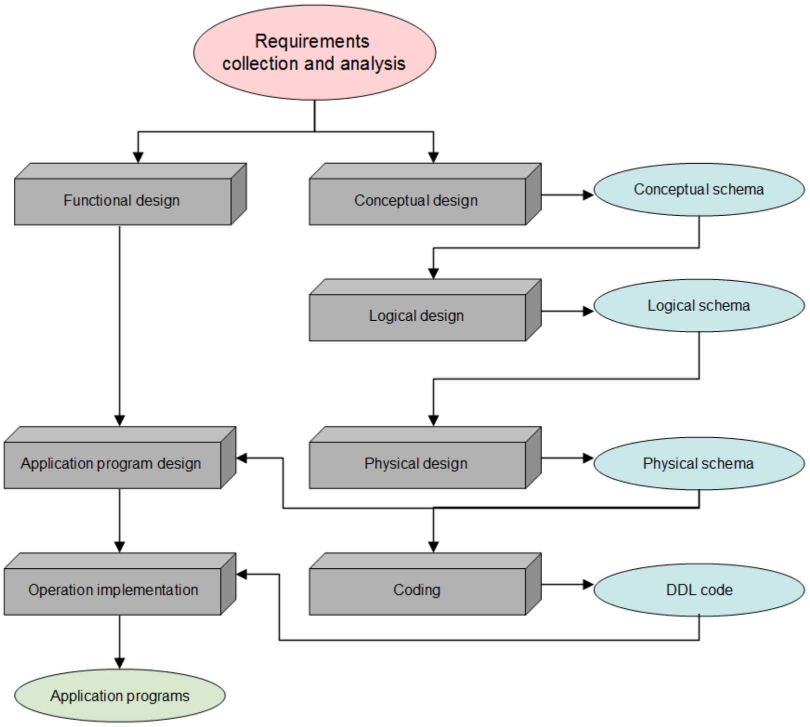


Figure 2.4: Database and application design processes

design processes. During these processes, different database schemas are produced, where each database schema represents a model, i.e., an abstract formal representation of a given application domain. Such a model allows one to better understand this application domain and to build an operational database that allows one to store and manipulate information about it (see Figure 2.4).

1. **Requirement collection and analysis:** The first step of this process is the collection and analysis of the users' requirements. This step seeks to gather and collect the user's requirements for the application domain under consideration. During this step, the database designers have to interview the users in order to understand what persistent data they want to store in the proposed system. The result of this phase is a document including the detailed requirements provided by the users in order to validate the understanding of requirements with users.

2. **Conceptual design:** The second phase, called conceptual design, also known as conceptual analysis, is where the database designers analyze the requirements document in order to produce the conceptual schema.

This latter should give a concise summary of all user's requirements in terms of what the data items are, what attributes they have, which constraints apply and what relationships hold between data items, as the overall aim is to develop a single shared database. The schema should not be overly formal or highly encoded but rather expressed in natural language.

3. **Logical design:** The third phase, called logical design, consists of translating the conceptual schema obtained in the previous phase into a logical schema. The logical schema is more concrete than the conceptual schema and it is more conceptual and abstract than the physical design. This phase attempts to describe concretely the database's structures, the relationships between these structures and the integrity constraints.

4. **Physical design:** The physical design consists of extracting the physical schema from the logical schema obtained in the previous phase. The physical schema is a representation of a data design as implemented, or intended to be implemented, in a Database Management System. The schema should include all the database artifacts required to create relationships between tables or to achieve performance goals (e.g. indexes, constraint definitions and linking tables).

5. **Coding phase:** The last phase is the coding phase that involves translating physical schema into Data Definition Language code for a database management system. The generated code allows one to create tables and constraints (e.g. checks and triggers).

2.6 Database Reverse Engineering

During the software system life-cycle, developers are often led to make changes not only in the program source code but also in the structures and constraints of their databases in order to adapt the database to ever-changing needs. As a consequence, such changes directly affect the current documentation when it becomes outdated. In addition, many legacy systems, including databases, have not been designed in a systematic way. In other words, they did not follow the software engineering process and the database engineering process during the design phase, which may result in a lack of documentation on programs and databases. Sometimes, the source code of the system and the DDL code of the database constitute the only available documentation.

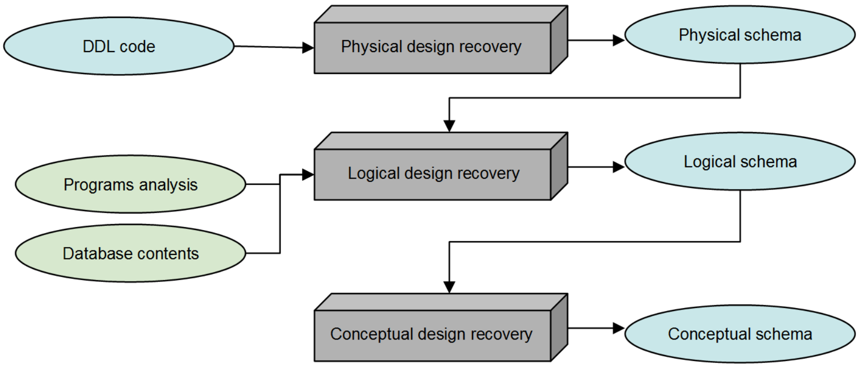


Figure 2.5: Database reverse engineering design processes

Recovering this information is an inevitable task when maintaining or extending old applications. To this end, the use of reverse engineering techniques is a necessary requirement. Here, it involves reconstructing the functional and technical documentation of the application program and the logical and conceptual schemas of its database, starting mainly from the source code of programs and the DDL code. As in the previous section, we just focus on the database reverse engineering design phases. The database reverse engineering is treated as the recovery of the conceptual schema starting from the DDL code. This process is depicted in Figure 2.5. The database reverse engineering process is composed of three successive steps. These are:

1. **Physical design recovery:** Starting from the DDL code of the database, we recover the physical schema. It is worth noticing that the physical schema obtained represents a raw schema, which means that it contains only the information belonging to the DDL code. Therefore, it is possible that some information linked to the implicit database constructs such as foreign keys will be overlooked.

2. **Logical design recovery:** This phase seeks to recover the logical schema from the physical schema obtained in the previous phase. To this end, we need additional information such as program source code and database contents to get a logical schema that is as complete as possible (e.g. to recover the foreign keys).

3. **Conceptual design recovery:** Lastly, in order to recover the most abstract schema, which is the conceptual schema, we used the logical schema to derive the conceptual schema.

2.7 The Generic Entity-Relationship Model

The basic Entity-Relationship Model (ER) is used to model a domain of application, which involves a transition from the real world to its computer representation. More specifically, the ER model is an abstract data model that defines a data or information structure that can be implemented in a database, typically a relational database. It was designed by Peter Chen [6].

The Generic Entity-Relationship model as defined by Hainaut [28] is an extended ER model, where it mostly comprises the concepts of schema, entity type, domain, attribute, relationship type, key, as well as various constraints. The GER model considers three levels of abstraction, these being: conceptual schema, logical schema and physical schema. We are now going to define the GER model according to two levels of abstraction (conceptual and logical schemas).

2.7.1 Conceptual Schema

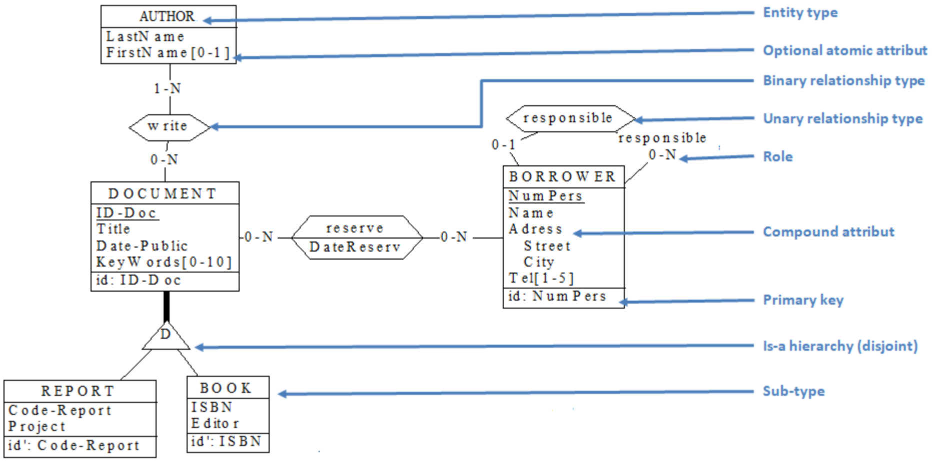


Figure 2.6: A conceptual schema example

A conceptual schema represents information gathered from business requirements, where entities and relationships modeled in such schema are defined around the business's need. In other words, a conceptual schema may be viewed as an abstract representation of the application domain (described by users). It seeks to delineate the specific entities in the software system, along with their attributes, and the relationships among these entities, where:

- Entity types are the core modeling constructs in an ER model. They represents the structure of top-level concepts, such as author, document or book. They can be organized into is-a hierarchies (super-types/subtypes). An is-a hierarchy may be ”total” (T), where a super-type may be specialized in at least one subtype or ”disjoint” (D), where a super-type may be specialized in at most one subtype or both ”partition” (P). Each entity type should contain the following information: a unique name, attributes which represent their properties, and a primary key defined by one or more attributes. Entity-types are related to other entity types through relationship types.

- Relationship types are used to associate one or more entity types. However, relationship types cannot exist as “stand alone”; there must be at least 2 entity types assigned to it. Like entity types, relationships may also have attributes. A relationship type is characterized by two or more roles, where each role has a cardinality constraint [i-n] that defines the possible number (minimum - maximum) of occurrences of one entity, which are associated with the number of occurrences in another. We call each relationship type that has exactly two roles a binary relationship type, while a relationship type with more than two roles is generally called an n-ary relationship type.

- Attributes are properties or characteristics of entities and/or relationships that hold it. An attribute has a name that describes the property and a type that describes the kind of attribute it is. It may be either atomic or compound. The compound attribute includes at least one sub-level attribute which may be atomic or compound.

Figure 2.6 shows an example of a conceptual schema using the generic entity-relationship representation. The schema describes a part of the application domain of a library management. It contains 5 entities (AUTHOR, DOCUMENT, REPORT, BOOK, BORROWER), each with its respective set of attributes (atomic or compound), linked through three association relationships: write, reserve and responsible. For instance, the author can write one or several documents, where each document is written by several authors.

2.7.2 Logical Schema

A logical schema is the translation of the conceptual schema into an operational schema complying with the data model of a particular database management system (DBMS), such as relational or object-oriented data model. The purpose of a logical schema is to describe how data should be organized and stored physically using a particular DBMS. In most cases, the logical schema is generated automatically from the conceptual schema, which consists of translating the conceptual objects into logical objects understandable by the target DBMS. For instance, a GER entity type will be translated to a table, while an attribute will be translated to a column using the relational terminology. Hence, the logical schema provides a lower-level description which is less intuitive in terms of understanding and more technical than the related conceptual schema.

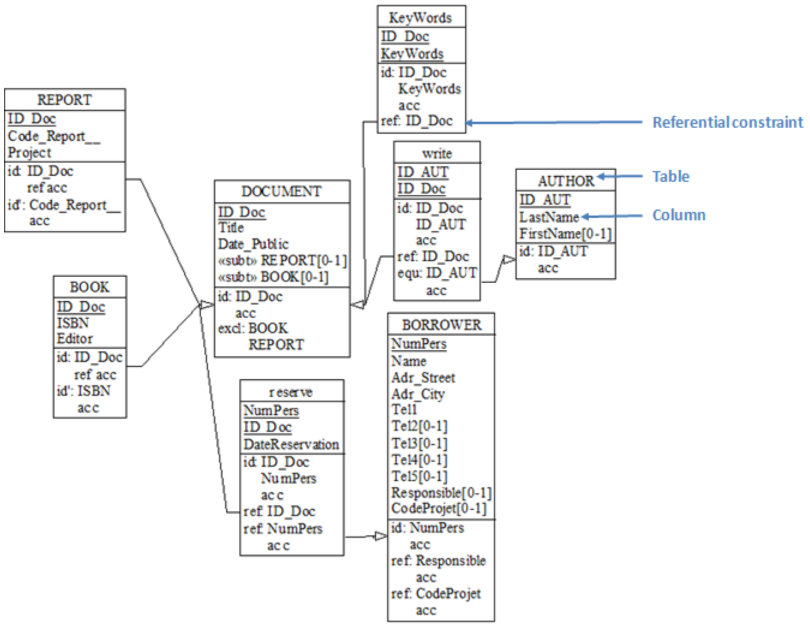


Figure 2.7: The logical schema example and rough relational translation of the conceptual schema of Figure 2.6

Figure 2.7 shows the logical schema of the conceptual schema described in Figure 2.6 after a transformation. The logical schema contains 8 tables (AUTHOR, DOCUMENT, REPORT, BOOK, BORROWER, KeyWords, write and reserve), each with its respective set of columns, where entities are translated into tables, attributes into columns and association relationships into either foreign keys or tables.

2.8 Program Analysis

Program analysis consists of developing algorithms and tools that can analyze other programs. It can be used to provide a result to developers (e.g. bugs and mistakes), to users (e.g. support for program comprehension), and also

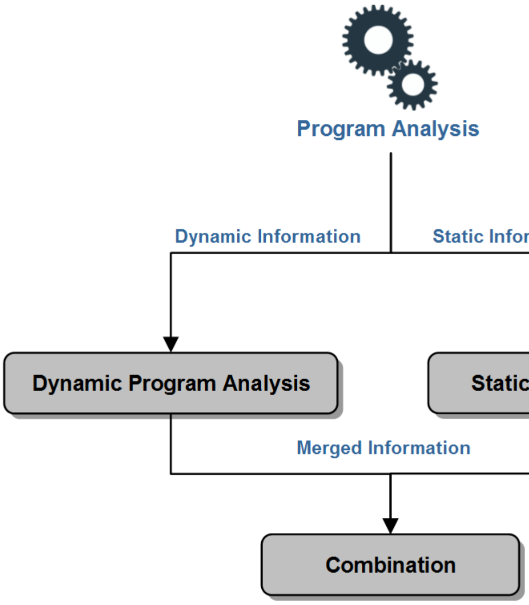


Figure 2.8: Programs analysis techniques

for other software tools, such as a compiler and optimizers. In the literature, there are several techniques available that can be grouped into two general categories (see Figure 2.8), these being static program analysis and dynamic program analysis.

2.8.1 Static Program Analysis

Static analysis is the analysis of programs without executing them, which is performed after coding and before running the programs. In other words, static program analysis includes a set of formal methods that seeks to extract information about the system behavior in order to understand it. Their main characteristic is the analysis of the program's behavior without execution. Those techniques are generally carried out either on source code or some form of the object code. In most cases, the analysis consists of exploiting source code and documentation artifacts. These techniques are generally used to optimize compilers in order to produce efficient code, detect bugs, ensure conformance to coding guidelines and others tools that assist program understanding. In the literature, there are several techniques available, but here we shall just mention some relevant ones:

1. **Hoare logic:** The Hoare logic technique was proposed by Hoare et al. [31] in 1969. This technique is a formal system with a set of logical

rules that attempts to work out the logical correctness of the programs. Hoare logic consists in a triple APB which asserts that *"If program P is started in a state satisfying condition A, if it terminates, it will terminate in a state satisfying condition B."*

2. **Abstract interpretation:** The abstract interpretation was formalized by Cousot et al. [17] in the late 1970. It is the formalization of the notion of approximation. The idea of an abstract interpretation is to create a new semantics of the programming language. In other words, it may be viewed as a partial execution of a program which seeks information about its semantics without performing all the calculations. The main contribution of such techniques is to understand why program static analyzers can be formally designed by discrete approximation of programming language semantics. This technique may be used to either analyze programs in order to decide whether certain optimizations are possible or apply transformations that are suitable for debugging.

3. **Data-flow analysis:** This was developed by Kildall et al. [34] in 1973. It is a static technique that tries to gather information about the possible set of values calculated at various points in a program. Starting from program's control flow graph (CFG), where the CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution [2], in order to determine those parts of a program to which a particular value assigned to a variable might propagate. This technique is often used by compilers to optimize the program.

4. **Model checking:** This began with the work proposed by Emerson et al.[21] in 1980. Model checking seeks to exhaustively and automatically check whether the model of a system given as an input meets a given specification. The aim of the model checking algorithm is to determine whether the abstraction of the program model satisfies the formula of program specification. The main contribution of such a technique is to be completely automatic and in the case where a property is not satisfied, the model returns a counterexample, and this led to a widespread use of this technique.

5. **Static program slicing:** The technique of slicing was defined by Weiser et al. [87] in 1981. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest, referred to as a slicing criterion (usually line-number and

variable), where program slicing is the computation of the set of program slices. The static program slice S consists of all statements in program P that may affect the value of variable V at some point of program P. This technique can be used for different purposes such as debugging, program comprehension, reverse engineering and program testing. There are many forms of slicing, which can be grouped into two main categories, these being static program slicing and dynamic program slicing, where the dynamic slicing can be thought of as an augmentation of the static form.

6. **Static code instrumentation:** According to Grossman [25], this is a technique that attempts to add additional code to a program/environment to monitor/change some program behavior. It is used for different purposes, such as software profiling, testing, debugging and optimizing. Code instrumentation can be performed with or without source code (using byte code or binary code). Static instrumentation without using the source code comes after compilation to add those additional machine instructions to the generated binary code. Then, the added instructions are executed with the others during the execution of the program.

A classic example using a static analysis technique is a compiler in order to find lexical, syntactic and even some semantic mistakes. For instance, static analysis is useful to detect formulas which use uninitialized or even undeclared variables. Moreover, several tools have been developed using static analysis to maintain code quality, program understanding and reverse engineering that we will present some of them in the next chapter. The well-known benefits of static approaches are mainly compactness and completeness. However, their results may contain some noise because they consider all possible execution paths of a program and variable values, by ignoring those invoked during a run-time execution.

2.8.2 Dynamic Program Analysis

In contrast to static analysis, the dynamic analysis of programs involves analyzing the program during runtime. More precisely, dynamic program analysis techniques focus on run-time aspects of programs by tracing their execution, and by considering a limited set of execution scenarios. These techniques potentially offer more details on important aspects such as late binding and accuracy. However, their results may overlook some program execution paths, thus leading to some silence. Another limitation of this technique is the potential impact that instrumentation may have on the execution of the program to be analyzed (including temporal properties). Therefore, in order to make the analysis more efficient, it is necessary to provide the target program with sufficient test inputs in order to produce significant behavior. In the literature, there are several techniques available, but here we shall just mention some relevant ones:

1. **Code coverage analysis:** This was used by Miller et al. in 1963 [52]. It is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. In other words, code coverage tests how much your code is covered under tests. This technique was invented mainly for systematic software testing.

2. **Dynamic program slicing:** In contrast with static program slicing, dynamic program slicing makes use of information about a particular execution of a program. According to Korel [36], it is characterized by the construction of a program slice which exploited the available information related to the input that caused the errors. Then, a dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program. This technique is useful for applications like debugging and testing.

3. **Runtime assertions:** The term assertion means a statement that is expected to be always true at the point where it appears in the code. If it is not (the assertion is evaluated false at runtime), an assertion failure results, which typically leads the program to crash, or to throw an assertion exception. This technique is generally used to help specify programs and to reason about program correctness. Taylor et al. [80] reviewed the use of assertions and they made suggestions for their incorporation in languages.

4. **Dynamic code instrumentation:** According to Nethercote [58], the dynamic instrumentation of a program is mainly based on two techniques. The first one is to instrument the byte-code of classes during loading phase. The second is to use the virtual machine in order to intercept only events that interest us using the hooks.

A well-known benefit of dynamic analysis is that we can detect subtle defects and vulnerabilities that cannot be detected by a static analysis. Although dynamic analysis can play a role in ensuring the security of software, its usual goal is to find and correct errors.

	Advantages	**Limitations**
Static analysis	- Reasons about all executions paths - Abstract domain (slow if precise) - Conservative due to abstraction	- Less precise - Sound (false positives and false negatives)
Dynamic analysis	- More precise - Concrete execution (slow if exhaustive)	- Results limited to observed executions - Unsound (silence) does not generalize

Table 2.1: Characteristics of static and dynamic analysis for program analysis

2.8.3 Static Analysis vs. Dynamic Analysis

So far we have presented two techniques belonging to program analysis. These techniques are normally used to assist developers in debugging, testing, program understanding, extracting the program's behavior, and so on. Both techniques provide different degrees of accuracy and completeness along with varying degrees of generality and compactness to deliver the expected benefits. We compared each one according to their advantages and their limitations. The results are presented in Table 2.1. We note that both are complementary because no single approach can meet all the criteria. This leads us to conclude that both techniques should never be thought of as if they were in direct competition with each other. Thus, it is more useful to consider them as mutually supporting techniques that together make the tasks of debugging, testing and program understanding easier and more efficient. Hence, there are numerous approaches available which use a combination of the two techniques in order to increase the effectiveness of the approach.

2.9 Software Visualization

In software engineering, there are two disciplines that concern visualization and are often confused with each other. These are: (1) *visual programming*, which is any programming language that uses graphical elements and figures to implement a program and (2) *program visualization*, which consists of producing animated views of information related to programs. Here, we will just focus on the program visualization that concerns the use of graphic representations in order to highlight certain aspects of software systems.

2.9.1 Definition

According to definition given by Roman et al [71]: ”*program visualization is the extraction of information about certain aspects of a program and their presentations in graphic form*”. A second definition is given by Petre et al. [67]: ” *Program visualization is trying to find simplicity in a complex artifact (e.g. several thousand lines of code), in order to produce a selective representation.*”. From these two definitions, we could deduct that software visualization is the static or animated 2D or 3D visual representation of information about software systems based on their structure, size, history or behavior, etc.

In recent years with the growth of data-intensive systems, they manipulate an ever-increasing amount of data. Software visualization became a new center of interest to support the understanding of such systems. In this context, several techniques have been proposed, where they focus on different aspects of software systems, such as source code, software structure, runtime behavior, component interaction and software evolution. These techniques can be grouped according to the classification given by Maletic in [47].

2.9.2 Classification of Software Visualization

In the literature, there are several types of visualization according to the diverse nature of these aspects. Figure 2.9 summarizes the different types of software visualization according to the classification proposed by Maletic [47].

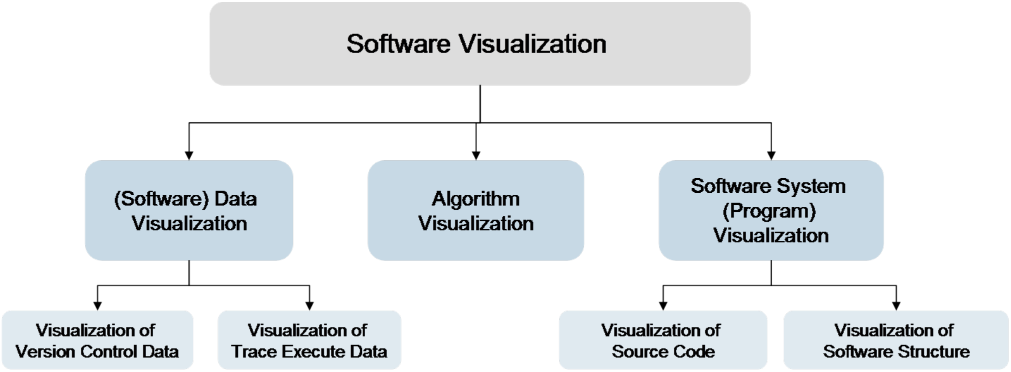

Software Visualization
(Software) Data Visualization
Algorithm Visualization
Software System (Program) Visualization
Visualization of Version Control Data
Visualization of Trace Execute Data
Visualization of Source Code
Visualization of Software Structure

Figure 2.9: Types of software visualization [47]

- **Program visualization:** Based on the definition given by Roman et al. [71], program visualization is a mapping from programs to graphical representations. And Richard Hamming [33] defined program visualization as the production of the animated views of program executions, where it is not limited to algorithms or activities that are evident in program source-code. It may include activities in the compiled code, the run-time system, data, and even the underlying hardware. It became very useful to support debugging, evaluating and improving program performance, evaluating and reducing resource utilization, evaluation of algorithms in the context of complete programs and real data, understanding program behavior and teaching. Recently, many tools have been proposed in this context. They differ widely in capability, technique, and applicability, and some of them will be presented in the next chapter.

- **Algorithm visualization:** According to Richard Hamming [33] definition, algorithm animation involves producing animated visualizations of algorithms, usually during program execution in order to evaluate and improve them.

- **Data visualization:** This is a general term that describes any effort to help people to understand the significance of data by placing it in a visual context. Patterns, trends, and correlations that might go undetected in text-based data can be exposed and recognized more easily with data visualization software.

2.9.3 How Do We Choose The Right Visualization?

As in any technique of visualization, we can raise one or several difficulties like getting the necessary data and information for visualization, identifying the aspect of program behavior to be visualized, finding a suitable visual representation for the behavior, considering the effects of the visualization on the behavior of the program being visualized and that the limited screen space often creates problems in presenting information from real programs, etc.

In the light of this, it is advisable to ask what a good visualization might be, what properties are needed to support and the characteristics it must have. Maletic et al. [48] define Quintilian questioning as that which allows researchers to ask the right questions in order to get a good delimitation of the purpose and also to choose from which perspective the goal will be treated. Then to know which kind of visualization is appropriate for the given purpose:

Criteria	**2D metaphor**	**3D metaphor**
Emergence	1950	1970
Popularity	more common	less common
Implementation	easier to implement	less easy to implement
Metaphors	graphs, trees, diagrams, etc.	geographic (countries, cities), graphs with 3 axes, etc.
Advantages	intuitive, simple, one piece of information at a time, etc.	several navigation modes, fairly clear overview, avoids overlap, etc.
Drawbacks	cognitive overload, restricted navigation	information overload

Table 2.2: Characteristics of 2D and 3D visualization

- **Why** is visualization needed?
- **Who** applies visualization?
- **What** should be represented?
- **How** should it be represented?
- **Where** should it be represented?

Once we have answered all these questions, the next step is to define the visual metrics and metaphors. According to the definition given by Diehl [19], a visual metaphor is an analogy that is the basis of a graphic representation of an entity or an abstract concept, where its aim is to transfer the properties of the graphic representation to those of the abstract entity or the concept. It is worth remarking that the choice of visual metaphor is crucial because it directly affects the effectiveness of the visualization. This is why it is necessary to be especially careful about the choice. Moreover, it is imperative that all visual metrics provided by the chosen metaphor are sufficient to represent all the required aspects; and above all, it can be employed consistently.

2.9.4 2D vs. 3D Visualization

In the literature, there are several types of visual metaphors available. According to the diverse nature of these metrics, we can classify them into two categories. These are:

1. ***2D visualization***: The main principle of this category is the use of graphics, trees and diagrams to visualize information that is given in a two dimensional representation.

2. ***3D visualization***: In contrast to 2D dimensional visualization, 3D visualization is a graphic that uses a three-dimensional representation to represent data.

Table 2.2 offers a small comparison between these two categories in terms of advantages, drawbacks, popularity and metaphors. Although 2D visualization is the oldest, it is the easiest to implement, the most widely used and intuitive. Nevertheless, it can quickly become overloaded, when there is too much information (data) to visualize.

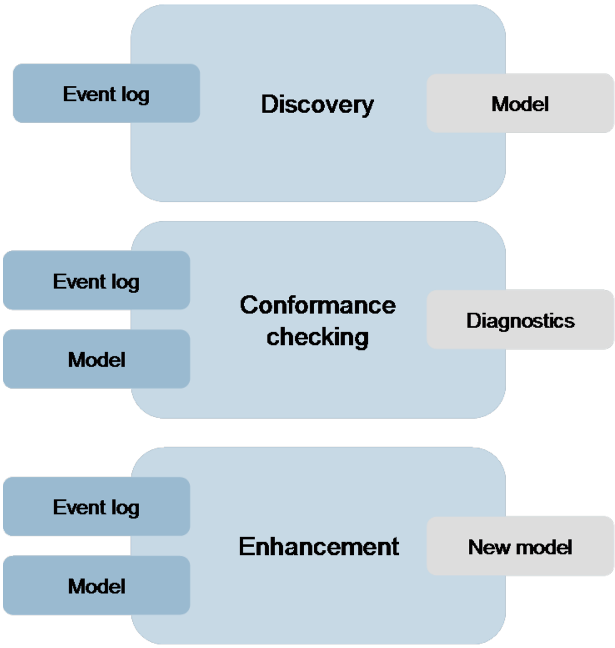


Figure 2.10: Types of process mining

2.10 Process Mining

Process Mining is a discipline that lies somewhere between process analysis and data analysis (machine learning, big data). According to the definitions proposed by [83] [84], process mining techniques seek to extract knowledge from event logs commonly available in today's information systems, in order to provide a new means to discover, monitor and improve processes in a variety of application domains. Therefore, the key ingredient of a process mining is an *event log*, which logs each step of the process to be analyzed. Starting from this input, the process mining can be divided into 3 axes [83]. Figure 2.10 summarizes them according to their input and output.

1. **Discovery**: It seeks to extract or discover a new process model from the low-level event log given as an input. This type improves the overall knowledge of the process based on its history. Many techniques were defined that sought to automatically extract process models (for example, Petri net [26], pi-calculus expression [75]) based on an event log.

2. **Conformance checking**: It seeks to compare the existing model and the process event log of the same process in order to check if the model of the process is conform with reality, as recorded in the log, conforms to the model and vice versa;

3. **Enhancement**: It attempts to extend or improve an existing process model with a new aspect or perspective using information about the current process recorded in some event log.

Process mining became a new focus of interest because :(1) an increasing amount of events is being recorded, and this provides a strong indication about the history of processes; and (2) there is a need to improve and support business processes in competitive and rapidly changing environments.

2.11 Conclusions

In this chapter, we presented different concepts and techniques related to our research domain that support software evolution and maintenance – more specifically the program comprehension of certain types of systems called "Data Intensive Systems" . We started by discussing software maintenance and evolution, then we focused on the first task, which is program comprehension because it is considered to be the most time-consuming task and this is especially true in the case of data-intensive systems. Afterwards, we defined different techniques that are mainly used in such areas and also which we will apply in our approach, namely database engineering/reverse engineering, static analysis, dynamic analysis, visualization and process mining.

In the next chapter, we will present some techniques and tools related to the understanding of data-intensive systems using the techniques described here.

Chapter 3

Related Literature

Great things are not done by impulse,
but by a series of small things brought together.
Vincent Van Gogh

Contents

3.1 Introduction 35

3.2 Program Analysis for Program Comprehension 36

3.3 Process Mining for Program Comprehension . . 44

3.4 Visualization for Program Comprehension . . . 46

3.5 Conclusions 49

3.1 Introduction

Many approaches have been proposed in the literature to assist program understanding. In this chapter, we review those that share a common goal which is *to help users and/or developers to better understand software systems*, or those who applied some techniques that inspired our approach such as program analysis, process mining and visualization. Below, we start by reviewing some of the existing approaches using *program analysis*, where we have grouped them into three categories: (1) approaches using dynamic analysis for program comprehension, (2) approaches using static SQL statement analysis. (3) approaches using dynamic SQL statement analysis. Then in Section 3.3, we review some existing approaches which make a use of *process mining* techniques to extract and understand the process models of systems.

In Section 3.4, we review existing approaches that use visualization for program understanding. Lastly, we present some shortcomings of the related approaches by identifying differences and bring out the main contribution of our approach.

3.2 Program Analysis for Program Comprehension

Many approaches have been proposed in the literature to assist program understanding. Program analysis has been considered as a valuable technique for a long time. It consists of supporting program-understanding tasks by analyzing a software system from its artifacts such as source code, documentation, architectural diagrams, design information, execution traces and the event log. In this section, we focus only on those that applied techniques that inspired our approach such as dynamic analysis, static and dynamic SQL execution trace analysis. However, only a few approaches focused on *data-intensive programs* - which is our main concern. Lastly, we synthesize and compare all contributions of these approaches according to our main contribution.

3.2.1 Dynamic Analysis for Program Comprehension

There are many approaches that concentrate on program comprehension through a dynamic program analysis. Cornelissen et al. [15] summarized most of them in a thorough survey, where they scanned a total of 4,795 articles that had been published between 1999 and 2008. Among those papers surveyed, they selected 172 articles that strongly emphasized the use of dynamic analysis in program comprehension contexts and 30 approaches that were related to the analysis of program execution traces. Out of these studies, we are only interested in those that are strongly related to our approach.

The approaches presented in [79, 3, 24] reveal the advantages of using a combination of static analysis and dynamic analysis for different purposes. The approach proposed by Systa et al.[79] introduces Shimba, a prototype of a reverse engineering environment, which combines static and dynamic analysis to help understand the run-time behavior of an object-oriented software system by analyzing the generated scenario diagrams. They employed a static analysis to select a set of components that need to be examined during a dynamic analysis. The approach is based on the assumption that a software engineer does not need to trace the whole system if only a specific part needs to be analyzed. Antoniol et al.[3] defined and implemented a WANDA

Year	**Authors**	**Technique**	**Systems**	**Objective**
2000	Systa et al.[79]	- Dynamic + static analysis - Visualization	Java software systems	Shimba - a reverse engineering environment to support the understanding of Java software systems
2004	Antoniol et al.[3]	- Dynamic + static analysis - Visualization	Web applications	WANDA - to recover the Web application architecture using the dynamic and static analysis
2006	Hamou-Lhadj et al.[29]	- Dynamic analysis	Object-oriented systems	Summarizing the content of large execution traces
2006	Greevy et al.[24]	- Dynamic analysis - Visualization	Object-oriented systems	Analyzing the evolution of systems through features views
2012	Trumper et al.[81]	- Dynamic analysis - Visualization	Embedded-systems	Software-based tracing technique for tracing and visualizing the runtime behavior of embedded systems
2013	Sarkar et al.[76]	- Dynamic analysis	Object-oriented systems	Understanding the dynamic behavior of object oriented systems
2013	Labiche et al.[39]	- Dynamic + static analysis	Java software systems	Reverse-engineer scenario diagrams for program comprehension
2018	**Noughi et al. [61]**	**- Dynamic analysis**	**Data-intensive systems**	**Understanding data manipulation behavior of data-intensive systems**

Table 3.1: Summary of research works focusing on program comprehension

(Web Applications Dynamic Analyzer) approach for the dynamic analysis of Web applications, where they combine the advantages of static analysis and dynamic analysis to recover the architecture of Web application; more specifically, UML documentation such as component, deployment, sequence and class diagrams. Greevy et al.[24] showed how a feature-centric analysis of a software system supported software evolution and maintenance activities. For this, they combined the dynamic models of feature behavior and the static models of the source code in order to extract the mapping between features and code, and then show that a features perspective of a system is an added-value to help understand the underlying reasons for changes.

In another context, Hamou-Lhadj et al. [29] presented a semi-automatic approach seeking to summarize the content of large execution traces in order to understand the main behavioral aspects of the given system using a dynamic analysis. The main objective of the approach is to take as input an execution trace in order to provide a summary of its main content by reducing its size and complexity, while retaining as much of its sense as possible. The traces they focused on were based on routine calls, which refer to any routine, function, or procedure, regardless of whether it is a method or a class.

The approaches presented in [81, 76, 39] make use of instrumentation for different purposes. The software-based tracing technique proposed by Trumper et al.[81] attempts to trace and visualize the runtime behavior of embedded software systems to support software maintenance; more specifically the program comprehension of such software systems. The authors make use of tracing in order to record the system runtime behavior, where they used a dynamic binary instrumentation technique to intercept the function entry and the function exit events. Then, they analyzed visually the resulting traces in order to provide a means for facilitating debugging and the program comprehension of embedded systems. Sarkar et al.[76] proposed an approach that sought to extract and understand the dynamic behavior of an object-oriented system through its UML sequence diagram. They used dynamic analysis in order to extract a sequence diagram from an object-oriented system program at runtime. For this purpose, they instrumented the Java code of the program and got a new instrumented Java code. Then, they generated a image file by a message sequence when they ran the instrumented Java code. Lastly, they got the sequence diagram from the image file. In the same context, Labiche et al.[39] combined static and dynamic analysis to reverse engineer the scenario diagrams of Java software, where the goal was to facilitate the software life-cycle activities; more precisely, program comprehension. The authors used static analysis to extract the control flow graph from Java software source code, and dynamic byte-code instrumentation in order to obtain program execution traces. Then, based on model transformations, they represented both sets of information using a UML sequence diagram representation.

3.2.1.1 Discussion and Comparison to Our Approach

From an analysis of the approaches in question, we noted that program understanding has existed for a long time and it had been the subject of several studies [79, 3, 24, 29, 81, 76, 39]. However, the problem of comprehension is still unresolved and it remains a key aspect of software evolution and maintenance.

Table 3.2.1 summarizes the research related to program comprehension through dynamic analysis or a combination of static and dynamic analysis. The first column recalls the year of the work. The second column includes the related work reference. The third column enumerates the techniques used. The fourth column highlights the targeted kind of software system. Lastly, the last column summarizes in one sentence the main objective of the proposed approach.

From an analysis of the approaches examined, we observe that none of the presented approaches focus on data-intensive systems (data-oriented systems), which have more problems than the comprehension of general-purpose systems. As we saw previously, these systems are characterized by intensive and complex interactions between programs and their databases, which are realized by SQL statements. Owing to this, our work will focus on the understanding of the data-manipulation behavior of data-intensive systems through an analysis of their SQL statements.

3.2.2 SQL Statement Analysis

Analyzing the data-manipulation behavior of programs via their SQL statements is one of the most powerful techniques available in software evolution and maintenance. Hence, extracting and analyzing the SQL statements of such systems may be viewed as a useful technique to understand, evolve or maintain them. This goal can be achieved in two different ways according to the capture technique of SQL statements used, involving a static analysis or dynamic analysis.

3.2.2.1 SQL Statement Static Analysis

Now, we present and discuss approaches that used static analysis in the context of SQL statement analysis. The studies outlined in [66, 92, 10, 82] proposed approaches based on an analysis of the access of SQL statements using static analysis for different purposes.

The technique presented in [66] seeks to extract an entity relationship (EER) schema from an operational relational database to improve database reverse engineering. The enrichment of the raw schema comes from an analysis of the SQL queries available in the application programs. In particular, joins are seen as heuristics for the detection of implicit dependencies between the columns of distinct tables.

Willmor et al.[92] proposed an approach based on a static program slicing technique. The authors defined an approach that supports slicing over both program and database state, where they introduced two new forms of states along with the standard program states in order to take into account the additional semantics of a database state. The first one is called a program-database dependency caused by the interaction between the program and its database state. The second is called a database-database dependency that exists between pairs of statements which both manipulate the database state.

In the same context, Cleve et al.[10] proposed an approach based on static analysis, namely a program slicing technique where they analyzed the source code of programs in order to detect and exploit data-flow dependencies that hold within and between (successive) SQL queries. They showed that the proposed approach, and its supporting tools, permit the recovery of implicit knowledge on the database structures and constraints such as undeclared foreign keys, finer-grained decomposition and more expressive names for tables and columns.

Vanden Brink et al.[82] presented an approach to the tool-assisted quality assessment of the data access aspects of systems that employ embedded SQL. The authors used a static analysis technique, more specifically a control and data-flow analysis, to reconstruct embedded SQL queries from the source code. Then, they analyzed them in order to define measures to quantify the quality of systems with embedded SQL queries.

Quite recently, some studies were interested in the static analysis of SQL queries, but for purposes other than program comprehension. For instance, Meurice et al.[51] defined a static analysis technique seeking to identify the dynamic database access locations and the database objects (tables and columns) accessed by a given access. They were interested in three types of Java database access technologies, these being Java DataBase Connectivity (JDBC), Hibernate and Java Persistence API (JPA). The authors identified the source code locations querying the database based on the call graph of the given method, where it is based on an inter-procedural analysis.

Later as an extension of the previous approach, Meurice et al.[50] proposed a tool-supported approach seeking to support the adaptation of application programs to database schema changes. To this end, they first analyzed how the source code and database schema had co-evolved in the past. Then, they simulated a database schema change and automatically determine the set of source code locations that would be affected by this change. Finally, they provided recommendations about what developers should modify at these source code locations in order to avoid inconsistencies.

In the context of a code smell detector, Nagy et al.[56, 57] defined a new approach seeking to help developers to accomplish SQL-related maintenance tasks in the code such as where it accesses a given part of the database, or which is responsible for the construction of a given SQL query. To this end, they performed various analyses on embedded SQL queries, once they had extracted them from JAVA applications using static analysis techniques. They also implemented the approach as an integrated eclipse plug-in called SQLInspect.

3.2.2.2 SQL Statement Dynamic Analysis

Although the previously studied approaches can extract to a certain extent the data access logic, the static program analysis techniques are limited in the case of highly dynamic database interactions, or in the presence of object-relational mapping technologies, involving the automatic generation of database queries at runtime. In such situations, dynamic information is crucial to attain a sufficient level of precision in the understanding process. The approaches presented in [18, 7, 1] analyze the access of SQL statements using dynamic analysis.

Del Grosso et al.[18] proposed an approach that attempted to automatically identify the application features of data-intensive programs in order to export these features as services. The method consists of collecting all SQL queries resulting from the interactions between the application programs and their database. Then, the authors applied clustering techniques in order to group together the SQL queries that access to the same database schema elements. Lastly, using existing migration techniques, they exposed them as a service.

The approach proposed by Cleve et al. [7] focused on the issue of database and program evolution; more precisely, the recovering of missing documentation about the database. The authors think that database re-documentation is an important source of information that is needed before evolving program and database. The main objective of the approach involves recovering the implicit database constructs and constraints, such as detecting implicit foreign keys of the database. With this in mind, the authors were led to analyze not only database code, but also other artifacts such as the source code of programs accessing to the target database via SQL execution statements using a dynamic analysis technique.

In a slightly different context, Alalfi et al.[1] presented the WAFA approach (Web Application Fine-grained Analysis), which analyzes the database interactions of dynamic Web applications in order to extract a fine-grained model and then to recover and control security properties such as the details of user roles and permission.

3.2.2.3 Dicussion

Many approaches have been proposed to assist software maintenance and evolution via an analysis of SQL execution traces. Here, we placed them into two categories according to the type of technique used: (1) approaches that focus on a static analysis of SQL execution statements [82, 92, 10, 66, 51, 50, 56, 57]; and (2) approaches that focus on a dynamic analysis of the SQL execution statements [18, 7, 1].

Table 3.2 summarizes all of them according to five aspects. The first column recalls the year of the work. The second column includes the related work reference. The third and the fourth columns define the technique used. Lastly, the last column summarizes in one sentence the main objective of the proposed approach.

Of course, these approaches demonstrated their relevance to software maintenance and evolution. However, we see that they have some limitations as well as certain aspects that were not considered:

- We can clearly state that very few of the given approaches focus on modern data-intensive (data-oriented) systems, i.e., applications where the most relevant features consist of interaction with different portions of a database. For such applications, it is necessary to recover the data-manipulation behavior in order to support the understanting.

- The approaches that used classical static analysis techniques are not sufficient in the context of highly dynamic systems. They generally fail to produce meaningful behavioral models in the case of these systems, which are mainly characterized by complex, dynamic and continuous interactions among the application programs and their databases. In this case, we observe that is more judicious to use dynamic analysis techniques to achieve more completeness and precision.

- We also observe that there is no approach supporting the analysis of program-database interactions with the ultimate goal of recovering and

Year	**Authors**	**Static**	**Dynamic**	**Systems**	**Objective**
1994	Petit et al.[66]	✓		Systems using relational database	Extracting an EER schema from an operational relational database to improve database reverse engineering
2004	Willmor et al.[92]	✓			Supporting the software maintenance process through program slicing
2006	Cleve et al.[10]	✓		Data-intensive systems	Recovering implicit knowledge on the database structures and constraints through program static slicing
2007	Del Grosso et al.[18]		✓	Data-intensive programs	Identify services in data-intensive programs
2007	Vanden et al.[82]	✓		Systems using embedded SQL queries	Quality assessment of systems through embedded SQL queries
2008	Cleve et al.[7]		✓	Data-intensive applications	Recovering implicit knowledge on the database structures and constraints using dynamic analysis technique
2009	Alalfi et al.[1]		✓	Web Applications	WAFA – An approach for Web Application fine-grained analysis
2016	Meurice et al.[51]	✓		Java Systems	Identifying the dynamic database access locations in Java Systems using static analysis techniques
2016	Meurice et al.[50]	✓		Data-intensive applications	Detecting and Preventing Program Inconsistencies under Database Schema Evolution
2017	Nagy et al.[56, 57]	✓		Java Systems	SQLInspect for statically extracting and analyzing SQL queries embedded in Java to support SQL-related maintenance tasks.
2018	**Noughi et al.[61, 60]**		✓	**Data-intensive systems**	**Understanding the data-manipulation behavior of data-intensive systems**

Table 3.2: Summary of research works focusing on SQL statement analysis

understanding the data-manipulation behavior of data-intensive programs. With the highly dynamic program-database interactions, the database queries may only exist at runtime. This is one reason that motivated us to develop a new tool-supported framework allowing developers to gain a better understanding of the data-manipulation behavior of data-intensive software systems. Our approach is able to analyze program-database interactions in order to understand data manipulation behavior of programs.

3.3 Process Mining for Program Comprehension

Process mining techniques can be seen as the fact of using methods of data analysis on problems related to a process in order to automatically discover process models, check the conformance of process models to reality, and extend or improve process models. These techniques may be regarded as useful and versatile in the context of program comprehension, where numerous approaches and tools have been proposed to assist program understanding using these techniques.

After examining the nature of data intensive systems, process models expressing data-manipulation behavior became a center of interest for developers, administrators and users. On one hand, developers and administrators (server-side users) design, develop, maintain and migrate data-intensive applications, thus supporting the whole system life-cycle. For migration purposes, they query models to capture the variability of the data-intensive applications in terms of the use of different functionalities (with data). Moreover, they recover sub-processes of the legacy system for understanding and checking their conformance with design-time models. On the other hand, users (client-side) should have access to the subset of the database that is required to perform the currently required data-manipulation functions. Hence, models expressing data-usage patterns for a single user can support his reconfiguration of data, e.g., by identifying data portions that it always accesses together.

The approach presented by Yang et al.[94] recovered an initial domain feature model from multiple existing domain applications to support the recovery of a feature model by means of an aspect-based tracing technique and formal concept analysis. The authors started by identifying the mapping between application data schemas and domain data model. Then, they used the data access semantics defined on database schemas as the business intent of all the program methods, where they identified a common basis of analysis for domain model recovery with consistent data access semantics. The authors applied formal concept analysis where they combined methods taken from different applications as objects and data access semantics as properties in order to extract the feature model.

The approach presented in [59] proposed a solution to the problem of identifying relevant processes from event log traces that capture Web service interactions. Information gathered from message interactions are analyzed to support the user in filtering in a semi-automatic manner the events of interest. Based on these logs, their approach is able to produce the corresponding process models that are relevant for the user.

In another context, Eisenbarth et al.[20] defined a new technique seeking to derive the feature-component correspondence, which represents a documentation artifact that describes which components are needed to implement a particular feature or set of features. With this in mind, the authors used dynamic information (got from execution traces, where they used a profiler to generate them) and concept analysis in order to get information on relationships among features and required components.

The approaches presented in [49, 73, 70] show how process mining can be applied to different application scenarios with the intention of understanding, checking and enhancing process models in real working environments. Mans et al. [49] demonstrated the applicability of process mining techniques to the health-care domain - more precisely, a gynecological oncology process. The authors analyzed the health-care process from three different perspectives, namely the control flow, the organizational perspective and the performance perspective, where they extracted different event logs from hospital information systems and analyzed them using the ProM framework (an extensible framework that supports a wide variety of process mining techniques in the form of plug-ins). The study outlined in [73] demonstrated the applicability of process mining techniques to less structured processes such processes of ASML (the leading manufacturer of wafer scanners in the world). The authors used the ProM framework in order to improve the testing process. In the same context, Rebuge et al. [70] utilized the techniques of process mining in the context of health-care processes in order to increase organizational performance by identifying regular behavior, process variants and exceptional medical cases.

3.3.1 Synthesis

Most of the approaches presented previously apply process-mining techniques to traces of system execution, but they do not consider data accesses. In addition, none of these approaches is suitable for data-intensive systems, i.e., systems where the most relevant features consists of interactions with different portions of a data sources. For such systems it is necessary to recover the data-manipulation behavior to support understating, checking and enhancement of their process models, which is our main concern.

3.4 Visualization for Program Comprehension

Program understanding is not an easy task, especially in the case of data-intensive systems, where it generally involves combining several techniques. Moreover, it is difficult to analyze, understand and manage a huge volume of data. Thus, it is necessary to present them in an understandable form. In the light of this, *visualization* is regarded as the most popular technique to resolve this problem. Hence, according to Stasko [78], all types of visualization seek to transform information into a meaningful, useful visual representation from which a human observer can gain an understanding. Visualization also has a big potential and it can lead to a better and quicker understanding of systems, which helps developers to save time and provide valuable information to help maintain and evolve them.

The use of advanced visualization techniques such as graphical visualization becomes even more indispensable in the case of software systems like which may include thousands of entities and rely on highly complex and dynamic interactions. A survey of the most popular visualization techniques was carried out by Caserta et al. [5], who summarized the most relevant techniques and tools in a systematic survey. They considered 2D and 3D based visualization, representing static aspects of the software and its evolution. They categorized and compared them to identify what was the most relevant for program comprehension.

In this section, we will summarize only those that somehow seem to be related to our approach. The approaches proposed in [32, 63, 14, 16, 40] were implemented in a 2D visualization tool.

Jerding et al.[32] proposed an approach seeking to identify, visualize and analyze interactions in object-oriented program executions in order to examine and understand dynamic program behavior. Their approach was implemented in an integrated tool called ISVis, which supports the analysis of the execution traces generated from object-oriented systems. The authors considered that large execution traces consist of recurring patterns, and visualizing these patterns is useful for reverse engineering. To this end, they defined a data structure for the internal representation of the traces, where they relied on the idea that a trace of method calls, which is a tree structure, can be transformed into its compact form. As a result, they obtained an ordered directed acyclic graph, where the same subtrees are represented only once. This representation allows ISVis to scale up to very large traces.

The approach proposed by Pauw et al.[63]] sought to visually explore Java program's run-time behavior visually. They implemented Jinsight – a visualization tool supported by the approach that is based on 2D visualization. It provided several views, that were linked to each other in many ways allowing navigation from one view to another. The authors affirmed that navigation makes the collection of views far more powerful than the sum of their individual strengths. Jinsight is used to help developers to find and fix bugs, for a performance analysis and any task where they need to better understand what the Java program is really doing.

Cornelissen et al. [14, 16] presented EXTRAVIS, a tool-based approach consisting of analyzing and visualizing the execution traces of programs in the context of trace understanding. They thought that the execution traces contained huge amounts of data that are not easily understood by the traditional visualization techniques. For this purpose, EXTRAVIS allows one to visualize, on one hand, these execution traces, and on the other hand, the program's package decomposition. In addition, it provides the means to navigate this information according to two synchronized interactive views, based on 2D visualization. These are: (1) EXTRAVIS' massive sequence view, consisting of a large-scale UML sequence diagram, that provides an interactive overview of the trace; (2) EXTRAVIS' circular bundle view, that hierarchically projects the program's structural entities on a circle and shows their interrelationships in a bundled fashion.

Lanza et al. [40] presented the Evolution Matrix, an approach seeking to visualize the evolution of classes in object oriented software systems. They combined software visualization with software metrics in order to help the recovery of the evolution of the object oriented software systems. The tool supported by the approach is based on a 2D visualization, which displays the evolution of the classes of a software system in the form of a matrix. Each column represents a version of software, while each row represents the different versions of the same class. In addition, the authors defined a set of metrics in order to reflect the measurements of classes. For instance, they used the metrics number of methods for the width and number of instance variables for the height.

In contrast to the approaches and tools defined above, the approach presented By Wettel et al. [88, 91, 90] was implemented in a language-independent interactive 3D visualization tool called CodeCity. The main principle of this approach is the analysis of large-scale object-oriented software systems using a city metaphor. Here, the authors represented the given system as a city, where classes are depicted as buildings (the number of methods represents the building height and the number of attributes represents the base size of the building), and packages as districts of the city. Such visualization allows one to get a preliminary understanding of the system in question, and gather some statistics about the program, which will enable users to identify the most sensitive classes and packages that are crucial for comprehension and maintenance.

Year	Approach	2D	3D	Tool	Objective
1997	Jerding et al.[32]	✓		ISVis	Analyze the execution traces generated from object oriented systems in order to detect the recurring patterns
2001	Pauw et al.[63]	✓		Jinsight	Explore Java program's run-time behavior visually
2001	Lanza et al.[40]	✓		Evolution Matrix	Gain a quick understanding of the evolution of classes within a software system
2007	Wettel et al.[88, 91, 90]		✓	CodeCity	A language-independent interactive 3D visualization tool for the analysis of large-scale object-oriented software systems
2009	Cornelissen et al.[14, 16]	✓		EXTRAVIS	A tool-based approach for the analysis and the visualization of the execution traces
2018	**Noughi et al.[62]**	✓		**DAViS**	**A tool-supported approach for dynamic analysis and visualizarion of SQL execution traces**

Table 3.3: Summary of research studies focusing on visualization

3.4.1 Synthesis

Numerous visualization tools have been proposed to assist program understanding. Here, we choose the most relevant for our contribution or those who have inspired us to design our tool. Table 3.4 summarizes them according to the kind of the visualization, the main objective and the year of their implementation. Through this study, we observe that a visualization represents an important contribution in the context of program comprehension, and more specifically in the case of data-intensive systems. However, choosing the right visualization in order to represent the information in an understandable form is the most difficult task. This is especially true when we consider the huge amount of data and information that we need to represent. In this case, the choice becomes crucial, because we have to take into consideration two aspects that have an inverse correlation, these being: (1) the amount of information to represent using a different visual metric and (2) keeping the simplicity of the visualization in order to make it intuitive and easy to understand.

3.5 Conclusions

Many approaches have been proposed to enhance the software maintenance and evolution phases. More precisely, program comprehension has always been an essential part of software maintenance and evolution. In this context, different approaches have been proposed to improve this phase of the software life-cycle. In this chapter, we outlined different approaches that support program understanding, where we classified them into three categories according to the used techniques. These are: (1) approaches that used program analysis, where it is regarded as the most common and appropriate technique to support the understanding of software systems; (2) approaches that used process mining techniques and (3) approaches that used visualization. For each category, we presented and discussed the most relevant studies related to our research work or that helped us to define our approach.

The next part will present our approach in detail. In chapter 4, we present an overview of the global approach as well as the research questions that we wish to answer. Then, we describe each part of the framework in chapters 5, 6 and 7. In Chapter 5, we elaborate on how we can use an SQL execution trace to extract the data-manipulation behavior of program. In Chapter 6, we show how we can use multi-SQL execution traces to extract the data-manipulation processes of programs. Lastly, in chapter 7, we outline how we can use the data-manipulation behavior to re-document source code.

Part II

Dynamic Analysis of SQL Execution Traces for Data-Intensive Systems

Chapter 4

Overview of The Framework

Big things have small beginnings

Prometheus

Contents

4.1 Introduction 53

4.2 Motivation 54

4.3 Preliminaries 55

4.4 Problem Statement 57

4.5 Approach Definition 57

4.6 Conclusions 60

4.1 Introduction

In the previous chapter, we showed how much effort has been made to enhance the software evolution and maintenance phase - more specifically, the program comprehension task. However, many limitations still remain and prevent program comprehension from being the least complex task of the software life-cycle. In the light of this, we elaborate on how to combine program analysis, visualization and process mining to support program comprehension. To this end, we defined an integrated framework seeking to understand the data-manipulation behavior of data-intensive systems via an analysis of their SQL execution traces. To achieve this goal, we start by presenting the motivation that allowed us to define our problem statement. Then, we present some formal definitions of concepts that are related to our methodology. Afterwards, we will explicitly define the problem statement. Then we will describe, at an abstract level, the different objectives of the framework we propose, and these will be touched upon throughout this thesis.

4.2 Motivation

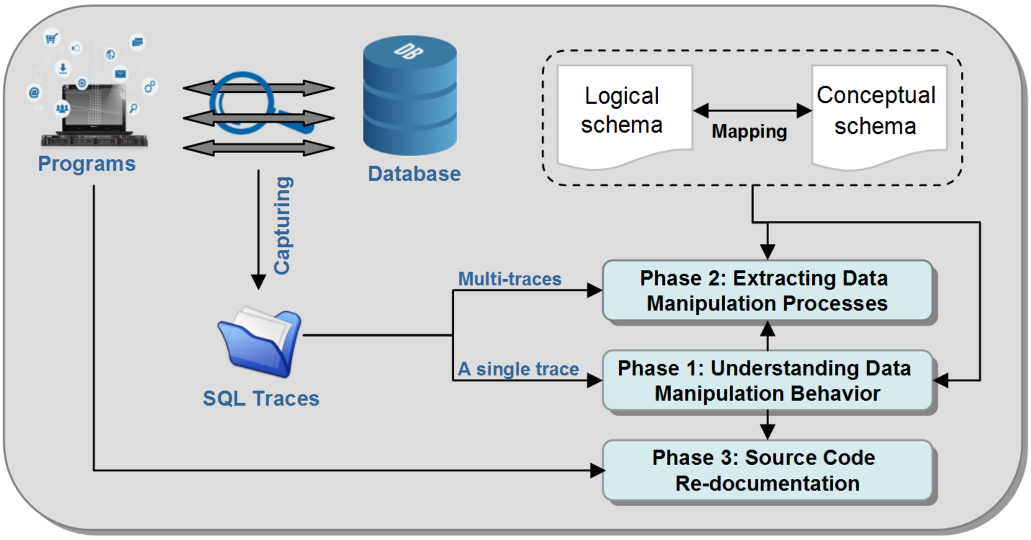

Programs
Capturing
Database
DB
Logical schema
Mapping
Conceptual schema
Multi-traces
A single trace
SQL Traces
Phase 2: Extracting Data Manipulation Processes
Phase 1: Understanding Data Manipulation Behavior
Phase 3: Source Code Re-documentation

Figure 4.1: General framework

As we said earlier, our main objective is to support the understanding of «*data-intensive systems*» by analyzing the communication between the application programs and their database, which is realized by SQL execution traces. For this purpose, we defined a framework which is summarized in Figure 4.1.

The framework includes three phases of analysis, where each phase has its own objective: (1) *phase 1*, seeks to support the understanding of data-manipulation behavior via an analysis of a single SQL execution trace; (2) *phase 2*, which seeks to recover data-manipulation processes via an analysis of multiple SQL execution traces and (3) *phase 3*, which seeks to re-document programs source code with their data-manipulation behavior. The framework was motivated by the fact that:

1. From the existing approaches so far, in the context of program comprehension, more precisely, those that were reviewed in Chapter 3, very few of them can be directly applied to modern data-intensive systems. We mean by data-intensive systems all systems that are mainly characterized by complex and dynamic interactions between the application

programs and a large set of data, through SQL queries, where the SQL queries are built dynamically at runtime.

2. Given the nature of these systems, program-database interactions constitute an important part of the program's behavior, which is important to consider and to understand. As luck would have it, none of the previously studied approaches (see Chapter 3) focuses on the analysis of these interactions for program comprehension.

3. In addition, the highly "dynamic" (Web applications, automatically generated queries, etc.) and "transparent" (Object Relational Mapping frameworks) nature of these interactions prevent them from being readily analyzed by means of classical program analysis techniques, such as static analysis.

4.3 Preliminaries

Now, we will introduce several concepts that will help to better understand the methodology we developed.

4.3.1 Mapping between Conceptual and Logical Schemas

The mapping between conceptual schema and logical schema can be defined as the correspondence relationship between each logical object with its corresponding conceptual object. In practice, each object belonging to the schema (logical or conceptual) has a meta-property (i.e., an annotation) called a Mapping Object Identifier (MappingOID). Hence, two objects belonging to distinct schemas (one to the logical schema and one to the conceptual schema) are mapped to each other if and only if they have the same Mapping Object Identifier (MappingOID).

Figure 4.2 shows the mapping between the conceptual schema (bottom figure) and the logical schema (down figure). The conceptual schema describes the application domain of a company selling products to customers. It contains 3, namely CUSTOMER, PRODUCT, ORDER, each with its respective set of attributes, linked through two association relationships (place and detail). For instance, customer can place one or several orders, where each order belongs to one and only one customer. And the logical schema represents the transformation of the conceptual schema above, where it contains 4 tables, namely CUSTOMER, PRODUCT, ORDERS and DETAIL, each with its respective set of columns, where entities are translated into tables, attributes into columns and association relationships into foreign keys.

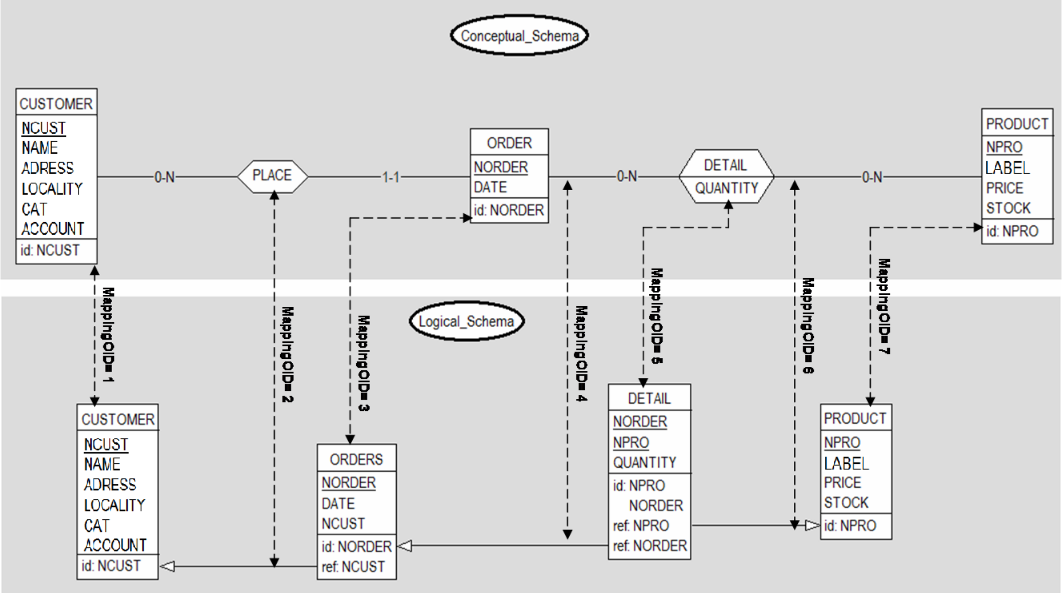


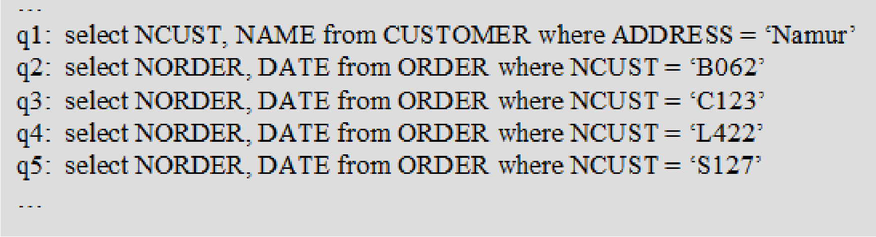
Figure 4.2: An example of a mapping between a logical schema and a conceptual schema.

For instance, the entity ”CUSTOMER” belonging to the conceptual schema is mapped with the table ”CUSTOMER” of the logical schema, where their *MappingOID* is equal to 1.

4.3.2 SQL Execution Traces

In software engineering, tracing means recording information about the program’s execution. Thus, a trace is a program execution representation. The tracing is used for different purposes, such as debugging and analyzing the program’s behavior according to the type and level of detail contained in the traces. Traces can be captured either statically, off-line, by analyzing the program source code or dynamically, at runtime, by capturing the generated code during the compilation and execution phases.

In the remainder of this thesis, traces represent the interactions occurring between application programs and their database, which are realized by SQL queries with their results. Hence, a trace is a sequence of SQL queries with their results. Through tracing, we are able to intercept all SQL queries that occurred in a program execution scenario with their results. From this, we obtain a sequence of queries with their results. Figure 4.3 depicts a fragment of an SQL trace, expressed in the input logical schema of Figure 4.2. The first query q1 searches the number and the name of all customers living in



```
...
q1: select NCUST, NAME from CUSTOMER where ADDRESS = 'Namur'
q2: select NORDER, DATE from ORDER where NCUST = 'B062'
q3: select NORDER, DATE from ORDER where NCUST = 'C123'
q4: select NORDER, DATE from ORDER where NCUST = 'L422'
q5: select NORDER, DATE from ORDER where NCUST = 'S127'
...
```

Result of q_1:

NCUST	NAME
B062	GOFFIN
C123	MERCIER
L422	FRANK
S127	VANDERKA

Figure 4.3: An example of a SQL execution trace + result of query q_1.

Namur. Each following query (q2-q5) selects the number and the date of all orders placed by a given customer.

4.4 Problem Statement

Given a set of SQL execution traces of the same program execution scenario, a logical schema, a conceptual schema and a mapping between them,

1. Can we automatically relate the program execution traces, the program source code and the database schema?
2. Does an analysis of the data-manipulation behavior support the understanding of data-intensive programs?
3. How can we automatically extract a model of the data-manipulation behavior?
4. Can we automatically re-document this behavior in the program source code?

4.5 Approach Definition

Starting from the key element of the problem statement, which is *«SQL execution traces»*, it is necessary to find a way to answer the above questions. To achieve this, a complete framework of the problem, research methodology and outputs are described in Figure 4.4. Using the different highlighted concepts, the framework includes three analysis phases, which are:

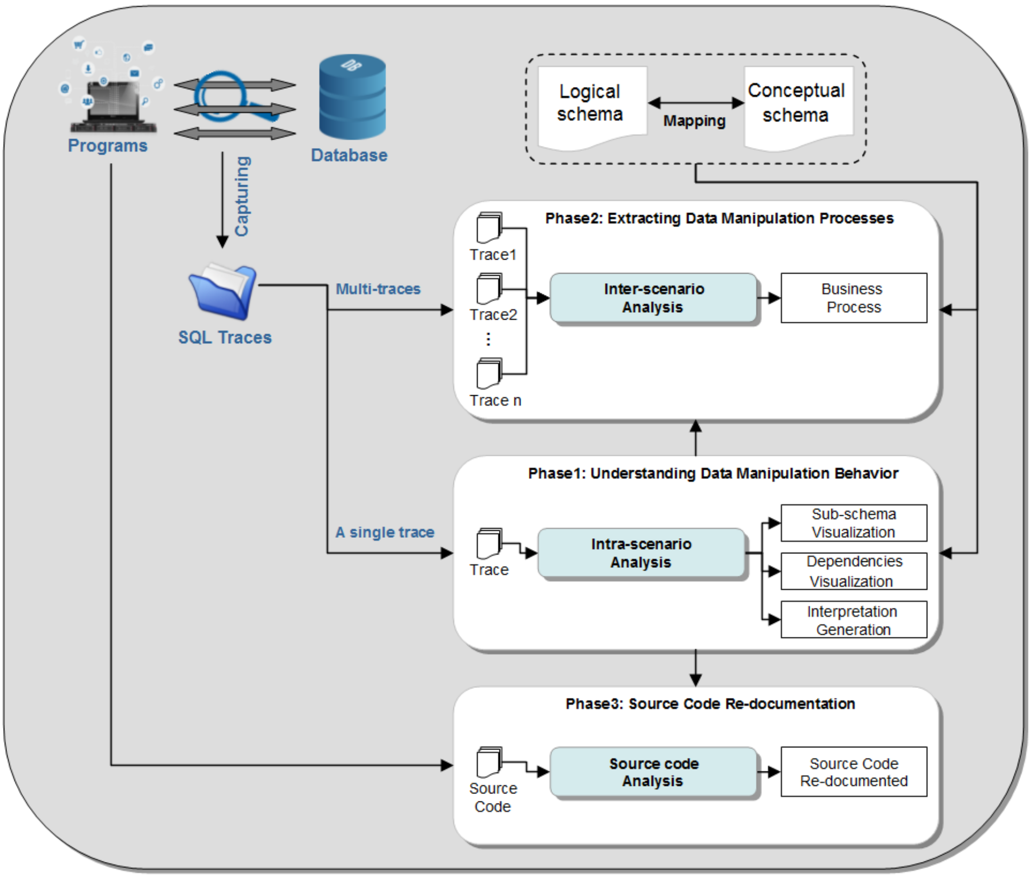


Figure 4.4: An overview of the framework

4.5.1 Phase 1: Intra-scenario Analysis

It starts from a single SQL execution trace that corresponds to a specific program execution scenario. This phase seeks to extract, analyze and understand the data-manipulation behavior of programs, in order to translate it into an easier representation that permits developers to better understand the impact of application programs on their database. For this purpose, based on dynamic analysis and visualization techniques, this phase provides three levels of understanding:

1. The first level is *the intra-query analysis*, which highlights the logical subschema affected by the trace given as an input and translates it into

an easier representation allowing an intuitive low-level understanding. In other words, it helps users to have a general overview describing the overall functioning of the program at the logical level.

2. The second level is *the inter-query analysis*, which consists of extracting and visualizing all implicit dependencies between queries belonging to the same SQL execution trace. This level seeks to help users to better understand how the successive queries depend on each other.

3. The third level is *the query interpretation*, the previous two levels have a significant contribution in terms of understanding logically the database-manipulation behavior of programs. However, they do not allow us to understand what the program really consists of from an abstract point of view. Based on the conceptual schema, this level tries to help users to better understand the data-manipulation behavior of programs by means of conceptual objects. To this end, it considers two kinds of representation. These are: (1) A conceptual visualization and (2) A natural language interpretation, which we will describe later.

4.5.2 Phase 2: Inter-scenario Analysis

The second phase of the approach seeks to recover the data-manipulation processes of data-intensive systems. The approach makes use of clustering, conceptualization and process mining techniques starting from multiple SQL execution traces corresponding to multiple (and possibly different) execution scenarios of the same «business process». What is more, the intra-scenario analysis phase (phase 1) constitutes the main basis of the inter-scenario analysis, insofar as it uses some of its outputs. The approach includes two steps, these being (1) *data-oriented properties*, where it is worth noting that this phase is a strictly database-oriented approach. Here, we consider queries implementing the same data-manipulation function as a unique event. Starting from these SQL traces, this step seeks to identify the data-manipulation function to which each query belongs; (2) *process mining*, which seeks to recover the data-manipulation process followed by the given SQL execution traces.

Moreover, we will discuss how we can exploit current practice technologies to implement the approach and we will carry out a set of experiments to assess the goodness of the recovered processes depending on the coverage of the input traces.

4.5.3 Phase 3: Code Re-documentation

According to Linares-Vásquez et al. [44], source code comments may be viewed as an another source of documentation that could help developers to understand nuances of the model and database usage. In the light of this, the third phase of our global framework is code re-documentation. The main objective of this phase is to re-document program source code with its data-manipulation behavior through comments. To achieve this, we defined a new approach that seeks to automatically generate a natural language interpretation of database-manipulation behavior in order to inject it as a comment into its corresponding source code location. More specifically, each method communicating with a database will be automatically commented based on the dynamic analysis results of SQL execution traces.

4.6 Conclusions

In this chapter, we began by formulating the overall objective of the thesis, which is ”understanding data-intensive systems by analyzing the interactions between application programs and their database, which are realized by SQL execution traces”. Then, we introduced the complete framework of the focused problem, research methodology and the expected outputs. This framework examined three phases of understanding, these being (1) intra-scenario analysis (phase 1), (2) inter-scenario analysis (phase 2) and (3) source code re-documentation (phase 3). The chapters following this one will elaborate each of these phases.

In Chapter 5, we describe the intra-scenario analysis, which supports the understanding of the database-manipulation behavior extracted from SQL execution traces. In Chapter 6, we describe the inter-scenario analysis, which supports the extraction of data-manipulation processes from SQL execution traces. Lastly, in Chapter 7, we outline the final part of the framework, which is source code re-documentation.

Chapter 5

Understanding the Data Manipulation Behavior from SQL Execution Traces

You may never know what results come of your actions,
but if you do nothing, there will be no results.

Mahatma Gandhi

Contents

5.1 Introduction 61

5.2 Approach and Research Questions 62

5.3 Trace Capturing 63

5.4 Intra-Query Analysis 69

5.5 Inter-Query Analysis 71

5.6 Query Interpretation 78

5.7 Tool Support 82

5.8 Case Studies 93

5.9 Conclusions 102

5.1 Introduction

In the previous chapter, we defined the global objectives of our framework divided into three parts of analysis. Now, we will detail the first analysis, namely an intra-scenario analysis. In this part will discuss how the analysis of program-database interactions could help developers to understand the data-manipulation behavior of data-intensive systems. First, we start by outlining the approach as well as the related research questions. Then, we present DAViS – a tool supported by the approach. Lastly, we introduce two case studies to illustrate and evaluate all the steps belonging to the defined approach through DAViS.

The work presented in this chapter was published in conference proceedings [8] [61] [60].

5.2 Approach and Research Questions

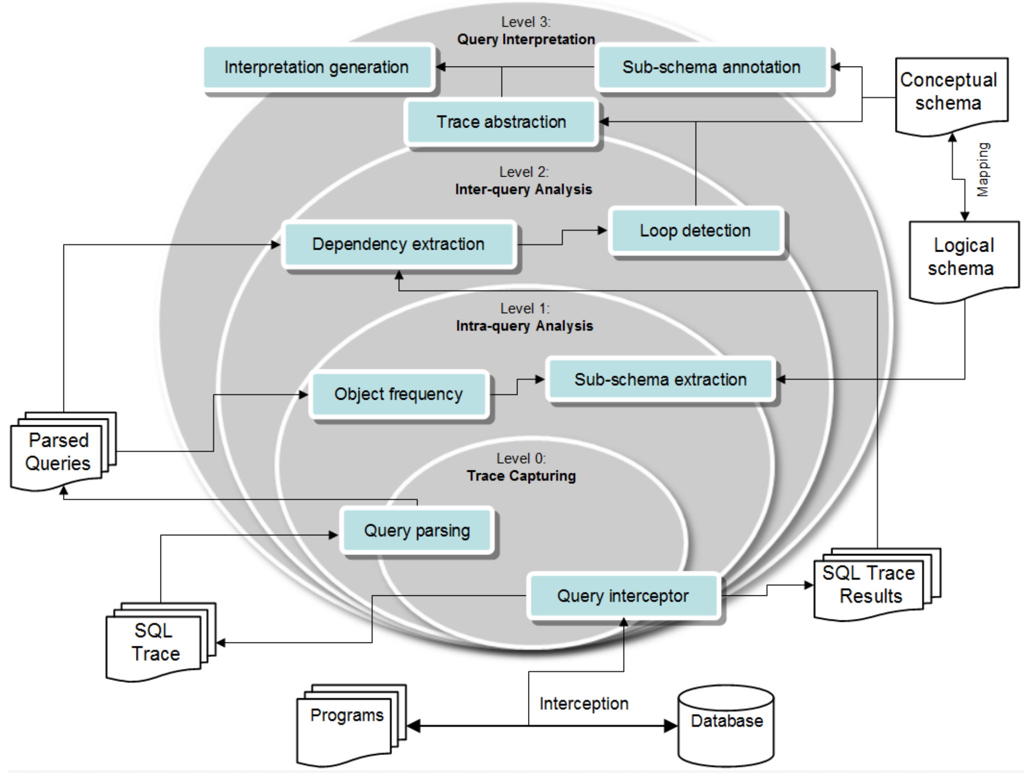


Figure 5.1: Bottom-up approach: Intra-scenario analysis.

Here, we are going to describe the approach that we have defined to assist the understanding of the data-manipulation behavior of data-intensive systems. The main idea behind our approach is that in data-intensive systems many interactions among application programs and database arise via database accesses. To this end, we combine the use of dynamic analysis technique and visualization in order to be able to identify, analyze and visualize these interactions. The approach considers four levels of understanding when analyzing a given data-intensive program execution scenario, each level requiring specific inputs. Figure 5.1 depicts the bottom-up approach used, starting from a single SQL execution trace:

- ***Level 0:(Trace capturing)*** is the initial level of the approach. This level attempts to answer *which database queries are executed in a scenario*. For this, first we identify the interactions between program and its database, which are realized by a set of SQL queries. Then, we parse them in order to extract the most relevant information to consider.

- ***Level 1:(Intra-query analysis)*** this level seeks to answer *which database schema elements are accessed by the SQL queries given as input*. Starting from the parsed queries, we highlight the subschema (logical and conceptual) affected by the given trace. Also, for each element belonging to the subschema, we compute its access frequency.

- ***Level 2:(Inter-query analysis)*** this level seeks to answer *how the successive queries depend on each other*. With this question, we extract the most relevant dependencies between successive queries belonging to the same SQL execution trace. For example: output/input dependency, input/input dependency and joint access dependency, which we will define later.

- ***Level 3:(Query interpretation)*** We look for the answer to the question about *what the SQL trace given as an input means in terms of domain specific concept manipulation*. Based on a conceptual schema, we interpret the trace according to a more abstract point of view, in terms of concepts and relationships (based on the conceptual schema).

5.3 Trace Capturing

As we mentioned previously, the interaction between the program and its database offers potential information about the data-manipulation behavior of a program. In this light, the trace capturing level seeks to capture all the interactions that occur between programs and their database. Indeed, these interactions are realized by SQL execution statements and their results. To achieve this objective, this level has two steps, namely: (1) query interception and (2) query parsing.

5.3.1 Query Interception

Many SQL traces intercepting techniques were presented by the authors of [8]. They range from simply using the DBMS log to instrumenting the application program via sophisticated program transformations. Among those techniques, let us briefly mention some of them:

1. **Database logs:** It is worth recalling that a database log is usually used to write all the data required to recover from a volatile storage failure and also from errors discovered by a transaction or database system. In other words, it is a history of actions executed by a database management system used to guarantee atomicity, consistency, isolation, and durability properties over crashes or hardware failures. For each action, a database management system records in a journal the following data: a start-of-transaction marker, the transaction identifier, the record information, the operation(s) performed on the records (select, insert, delete, modify), the previous values of the modified data, the update values of the modified records, a commit transaction marker if the transaction has committed or otherwise an abort or rollback transaction marker. For instance, MySQL writes in its general query log, all the executed queries in the order it received them. The main advantages of this technique are the fact that it is processed off-line and it does not require any modification nor recompilation of the program source code. However, it does not include the results of those queries nor any source code location information that allows one to map the executed queries to the program source code files. Furthermore, the log includes other information that is not interesting for reverse engineering purposes, such as the queries accessing the system tables.

2. **Tracing aspects:** Aspect-oriented programming (AOP) is a programming paradigm that seeks to increase modularity by allowing the separation of cross-cutting concerns. The main focus of this paradigm is an aspect, or feature that can be found across methods and classes. An aspect can be viewed as a combination of point-cut and advice. A point-cut represents the point of execution in the application at which cross-cutting concern needs to be applied and an advice is a piece of code that should execute at each of the join points specified by the point-cut. AOP is typically used in the context of logging. For instance, when we want to log a method calls, a method execution or an object instantiation.

3. **Program instrumentation:** As we defined in Chapter 2, static and dynamic program instrumentation attempts to add additional code in

the source code, byte-code or binary code. In the context of trace capturing, it is possible to use program instrumentation to capture SQL execution traces and their results. To this end, it is necessary to implement instrumentation in the form of code instructions that monitor the dedicated code section before each program point of this statement and after its result. This technique requires the analysis of program source code in order to identify the location of the executed SQL statements. However, the instrumentation is limited by execution coverage, which means that if a program never reaches a particular point of execution, then instrumentation at that point collects no data. In addition, it could cause a drastic increase in the program execution time.

The approach we defined is actually completely independent of the chosen SQL trace capturing technique, where it requires a minimal input as a sequence of SQL queries corresponding to a given program execution scenario, together with the results of these queries. However, according to the chosen technique, we obtain traces which contain different information. Hence, we implemented an algorithm that sought to produce a unified XML file attempting to represent any kind of SQL statement trace given as an input. In addition, the result of each query is recorded in a MySql database.

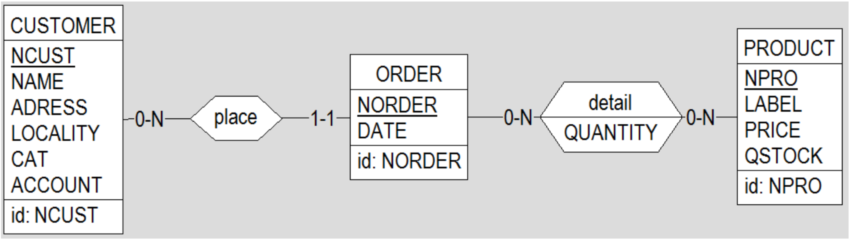


Figure 5.2: The conceptual schema of a Customer-Order application

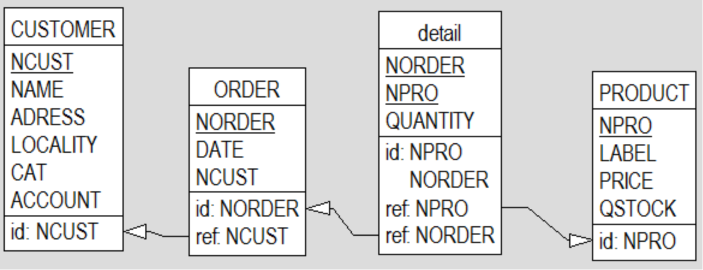


Figure 5.3: The logical schema of a Customer-Order application

```
q1: select name, ncust from customer where Locality = 'Namur';
q2: select norder, ncust, date from order where ncust= 'B062';
q3: select norder, ncust, date from order where ncust= 'C123';
q4: select norder, ncust, date from order where ncust= 'L422';
q5: select norder, ncust, date from order where ncust= 'S127';
q6: select name, ncust from customer where Locality = 'Poitiers';
q7: select norder, ncust, date from order where ncust= 'B112';
q8: select norder, ncust, date from order where ncust= 'C400';
q9: select norder, ncust, date from order where ncust= 'F010';
q10: select ncust, name from customer where ncust not in (select ncust
    from order);
...
```

Result of q_1:

NAME	NCUST
GOFFIN	B062
MERCIER	C123
FRANK	L422
VANDERKA	S127

Figure 5.4: An example of an SQL trace extracted from Customer-Order application + result of query q_1

In the case of the illustrative examples and the case studies, we used the database log to intercept SQL traces, where, we recorded the intercepted SQL queries into an unified XML file and their results in a MySql database. Each result is identified by the name of the scenario and the sequence number of the query.

To illustrate the different steps of the defined approach. Figure 5.2 depicts a «simplified» conceptual schema with its corresponding logical schema (Figure 5.3) describing the application domain of a company selling products to customers. The logical schema contains 4 tables, namely CUSTOMER, PRODUCT, ORDERS and detail, each with its respective set of columns. Let us now assume that a given execution scenario of a given application program accesses the information about customer orders. Through the query interceptor step, we are able to intercept the SQL queries involved in the scenario in question (Figure 5.4), where queries $q1$ and $q6$ select all *customers* living in $Namur$ and *Poitiers*, respectively. Then, ($q2$, $q3$, $q4$, $q5$) and ($q7$, $q8$, $q9$) select all *orders* placed by all the *customers* living in $Namur$ and *Poitiers* respectively. Finally, $q10$ select all customers who have not placed an order. And the results of each SQL query are recorded in a MySQL database. Each result is identified by the name of the scenario and the sequence number of the query.

5.3.2 Query Parsing

Once SQL execution traces have been collected, the query parsing step attempts to parse each query belonging to the trace in order to extract the most relevant information that will be the basis for understanding the data-manipulation behavior of the given data-intensive system. To this end, we implemented a parser that seeks to decompose each SQL query into a set of clauses. The parser takes one SQL query as an input, and according to the nature of the query, it produces a unified data object including clauses that can represent any kind of SQL statement (select, update, insert, delete, etc.).

To achieve this goal, we decided to utilize *JSqlParser* [1], which is a Java library, to parse statements according to the basic grammar of SQL. This grammar includes the common CRUD operations, namely *select*, *delete*, *insert*, *replace*, *update*. This parser analyzes and translates a SQL statement into a hierarchy of Java classes. Based on this hierarchy, we created a Java object that represents any query by just considering a *subset* of the information produced by the parser. We decomposed a query into several clauses containing different information according to the query type. In the following, we will describe the clauses that we recorded for each type of query. These are:

- **SELECT:** we record the *select* clause containing the output attributes, a *from* clause with the names of tables, and a *where* clause expressing the conditions for selecting the attributes of the select clause. These conditions may vary from a simple comparison operation to possible join conditions or a sub query. In the case of a simple comparison condition, we record the operands and the comparison operator; in the case of a join condition, we record the tables involved in the join along with their join conditions; while in the case of a sub-query condition, we iteratively record the same information as for the external select query. However, we add the query level information (for instance level 0 for the main query, level 1 for the first nested query).

- **DELETE:** we record the *delete* clause containing the name of the table from which delete rows and a *where-clause* as obtained for the where clause of the SELECT query.

- **UPDATE and REPLACE:** we record for such queries a clause containing the table name, the *set* clause containing all the attributes to edit or replace (of the form: column = value) and a *where* as presented for the previous queries.

[1] http://jsqlparser.sourceforge.net/

- **INSERT:** we record the *insert* clause with the table name, the *set* clause containing all the attributes to insert and the *values* clause containing the values for the corresponding attributes.

However, it should be noted that parser just considers the recovered or modified information and the related selection criteria, and it ignores all logical operators (and, or) between conditions (for instance in the where-clause).

Query	**Clauses**

SELECT name, ncust
FROM customer
WHERE locality = 'Namur';
	Selectclause={name, ncust}
	Fromclause={customer}
	Whereclause={(=, locality, 'Namur')}

SELECT ncust, name
FROM customer
WHERE cat = 'C1' and ncust not in (select ncust from order);
	Selectclause={(ncust, name), (ncust)}
	Fromclause={customer, order}
	Whereclause={(=, cat, 'C1'), (not in, ncust, select...) }

UPDATE order
SET norder = 32540
date = "2017-02-03",
ncust= "CS464"
	Updateclause={order}
	Setclause={(=, norder,32540),
	(=, date,"2017-02-03"),
	(=, ncust,"CS464")}

INSERT INTO order
(norder, date, ncust)
VALUES (32541,
"2017-02-04", "CS464")
	Insertclause={order}
	Valuesclause={(=, norder,32541),
	(=, date,"2017-02-04"),
	(=, ncust,"CS464")}

DELETE FROM customer
WHERE ncust ="CS464"
	Deleteclause={customer}
	Whereclause={(=, ncust,"CS464")}

Table 5.1: Example of the parser structure

Table 5.1 lists the parsing (in terms of clauses) of five examples of queries extracted from the collected traces of Customer-Order application (Figure 5.3). For instance, the first query depicts the parsing of a select query, where we obtained three clauses (SelectClause, FromClause and WhereClause), each of them containing a specific item of information.

5.4 Intra-Query Analysis

Starting from the data object including the parsed queries, this level attempts to analyze independently each query in order to extract all the statistics about data-manipulation behavior in a given program execution scenario.

5.4.1 Sub-schema Extraction:

The identification of database fragments involved (i.e., read or modified) in a given program execution scenario is the first level of understanding of data-manipulation behavior. This phase consists of filtering out from the input logical schema the elements that have not been accessed by any of the queries appearing in the input trace.

Definition 5.4.1. *Given a set of SQL queries belonging to one execution trace (T), the output is a subschema SubSch that must respect the following condition:*

$$SubSch = \{obj | \exists q \in T : obj \in q.objs\}$$

- T denotes one execution trace representing a sequence of SQL queries;
- q denotes an SQL query occurring in T;
- obj denotes the name of a table or a column;
- $q.objs$ denotes the set of objects used by q;

Starting from the set of queries belonging to one SQL execution trace T, we defined an algorithm (Algorithm 1) implementing the previous definition. The algorithm seeks to identify the sub-schema $SubSch$ affected by the trace T given as an input. The algorithm starts by parsing each SQL query q belonging to the trace T in order to extract all objects ($q.objs$) present in q (Line 3). Then, for each object belonging to $q.objs$, the algorithm makes the concerned object obj visible in $SubSch$ (Line 5).

Algorithm 1 Sub-schema extraction algorithm

```
Require: A logical schema of the system $LS$ and an SQL trace $T$
Ensure: Sub-schema $Subsch$ impacted by $T$
$SubSch \leftarrow LS$
for all $q \in T$ do
    $q.objs \leftarrow Parse(q)$
    for all $obj \in q.objs$ do
        $SubSch.SetVisible(obj)$
    end for
end for
```

5.4.2 Object Frequency

A parsed query can be viewed as a set of logical objects belonging to the logical schema and manipulated by the initial query, such as columns, tables and relationships between objects (e.g. join). However, knowing how many times each object is used in the same input trace of the same scenario may be regarded as an important item of information and this may help one to understand the impact of each object on the program as well as on the database. With this in mind, for each object (obj) belonging to the input trace (T), we attach a property called *Frequency (freq)*, which represents the number of queries of the same trace in which the current object (obj) is used, according to the following definition:

$$freq(obj, T) = |\{q \mid q \in T \wedge obj \in q.objs\}|$$

- *obj* denotes the name of a table or a column;
- $q.objs$ denotes the set of objects used by q;

Based on the previous definition, we defined an algorithm 2 that seeks to compute the access frequency of each object belonging to the trace given as an input. Starting from the set of queries in a trace T, the algorithm starts by parsing each query q belonging to the trace T (Line 2), and increments for each element its access frequency (Line 4). In the end, we obtain a set of objects with their access frequency property.

Let us now apply these two algorithms on the extracted SQL trace (Figure 5.4) from the described application program of Figure 5.3. We automatically obtained from the *intra-query analysis* the affected subschema described in Figure 5.5), with the corresponding access frequency of each object belonging to this sub-schema (Table 5.2).

Algorithm 2 Object frequency algorithm

```
Require: A sub-logical schema of the system $SubSch$ and an SQL trace $T$
Ensure: $obj.freq$ of each object belonging to $T$
for all $q \in T$ do
    $q.objs \leftarrow Parse(q)$
    for all $obj \in q.objs$ do
        $obj.freq \leftarrow obj.freq + 1$
    end for
end for
```

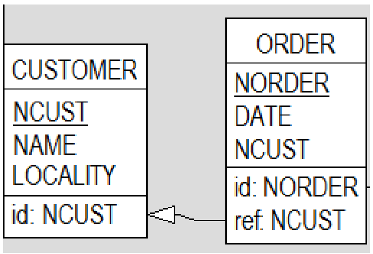


Figure 5.5: Sub-schema affected by the trace of Listing 5.4

5.4.3 Synthesis

We could conclude that the first level of the intra-scenario analysis, which is the intra-query analysis, attempts to highlight two main aspects of the data-manipulation behavior of programs. These are:

- Here, we just focus on logical objects of the database that are affected by the analyzed trace, where we have eliminated all the other objects from the logical schema given as input. This allows us (developers) to save time by avoiding the analysis of non-affected objects before making any changes on the application programs and/or database.

- In addition, the access frequency allows us (developers) to gain a better insight into the data-manipulation behavior by identifying the most frequent objects to be taken into consideration before making any changes on application programs and/or database.

5.5 Inter-Query Analysis

Starting from the parsed queries, this level seeks to determine how successive queries depend on each other and it has two steps, these being: (1) dependency extraction and (2) loop detection.

Objects	**# Frequency**
CUSTOMER	3
CUSTOMER.NCUST	3
CUSTOMER.NAME	3
CUSTOMER.LOCALITY	2
ORDER	8
ORDER.NORDER	7
ORDER.DATE	7
ORDER.NCUST	8

Table 5.2: The object frequency of the trace of Listing 5.4

5.5.1 Dependency Extraction

The *dependency extraction* step consists of implementing different algorithms that attempt to extract implicit links between queries involved in the same trace. More precisely, we are interested in three types of dependencies. These are: (1) *Joint access dependency*, (2) *Output/Input dependency* and (3) *Input/Input dependency.*

Definition 5.5.1. ***(Joint access dependency)*** *We identify a joint access dependency between two tables t_1 and t_2, when we have a query q belonging to the trace T which uses both t_1 and t_2. In formal terms:*

$$\vdash \exists q \in T : t_1 \in q.objs \land t_2 \in q.objs$$

- t_i denotes a table i of a relational database;
- $q.objs$ denotes the set of objects used by q;

Listing 5.1 shows an SQL execution trace fragment from the trace described in Listing 5.4 containing an example of a joint access dependency between two tables (*customer* and *order*). We notice that query $q10$ uses the table *customer* and the table *order*.

Listing 5.1: An SQL query extracted from the trace 5.4 with a joint access dependency

```
...
q10: select ncust, name from customer where ncust not in (select ncust
    from order);
...
```

Definition 5.5.2. ***(Output/Input dependency)*** *We identify an output/input dependency (O/I) between two successive queries q_i and q_j belonging to the same trace (T), when query q_j uses as input some of the results of q_i and query q_i is executed before q_j. In formal terms:*

$$\vdash \forall i, j \in [1, |T|] : (i < j) \Rightarrow (q_i.out \cap q_j.in) \neq \emptyset$$

- $q_j.in$ denotes the set of input values of q_j;
- $q_i.out$ denotes the set of output values of q_i;

The O/I dependencies can be identified in different cases. For instance, in a procedural join between the source and target tables of a foreign-key, the value of the foreign-key column(s) is used to retrieve the target row using a subsequent query. Conversely, the value of the identifier of a given target row can be used to extract all the rows referencing it.

Listing 5.2: An SQL execution trace fragment with an output-input dependency.

```
q1: select name, ncust from customer where Locality = ’Namur’;
// getString(1) = B062
q2: select norder, ncust, date from order where ncust= ’B062’;
...
```

Listing 5.2 depicts an SQL execution trace fragment containing an example of an output-input dependency between two successive queries (q1 and q2). We observe that the output of value of column *ncust* of the first query is the same as the input value of column *ncust* in the second query.

Definition 5.5.3. ***(Input/Input dependency)*** *We identify an input/input dependency (I/I) between two successive queries q_i and q_j belonging to the same trace (T), when q_i shares common input values with q_j. In formal terms,*

$$\vdash \forall i, j \in [1, |T|] : (i < j) \Rightarrow (q_i.in \cap q_j.in) \neq \emptyset$$

Listing 5.3: An SQL execution trace fragment with an input-input dependency.

```
q1: select count (*) from customer where ncust = ’B062’;
q2: insert into order (norder , DATE , NCUST )
values (47869 , ’28/02/2014 ’ , ’B062’)
...
```

As an illustration, we will examine the fragment of SQL execution trace given in Listing 5.3. The fragment includes two SQL queries extracted with an input-input dependency on the value *'B062'*.

The presence of input-input dependencies in SQL execution traces constitutes another strong indication of the presence of foreign keys [8]. Several data-manipulation patterns for referential constraint management make intensive use of input-input dependent queries. Among the most popular examples, the delete cascade mechanism, which involves deleting all referencing rows before deleting a target row, makes use of delete queries that share a common input value: the primary/foreign key value of the target rows to be deleted. A second example is the check-before-insert pattern, which attempts to preserve a referential integrity constraint when inserting rows in the database. When inserting a row in a referencing table, the program first checks that the provided foreign key value is valid, i.e., it corresponds to the primary key value of an existing row in the target table. Similar patterns can be observed in delete and update procedures.

5.5.2 Loop Detection

The inter-query analysis step also seeks to automatically detect potential loops. In other words, the *loop detection* step supports the identification of nested queries in the application program. We say that the queries $q_2, q_3, ..., q_n$ are nested in the query q_1, if their execution is performed before the result set of the preceding query q_1 has been completely emptied. Such a situation strongly suggests that a data dependency exists between the output values of q_1 and the input values of $q_2, q_3, ..., q_n$ or, in other words, $q_2, q_3, ..., q_n$ are most probably output-input dependent on q_1.

To achieve this aim, we implemented an algorithm that is summarized in Listing 3. The algorithm consists in analyzing the imbrication relationship between SQL statements, according to the following definition:

Definition 5.5.4. ***(Loop dependency)*** *We identify nested queries belonging to the same trace T, when:*

$$\vdash \forall i \in [1, |T|], \exists j \in]i, |T|] : \textit{(}q_i.out \cap q_j.in\textit{)} \neq \emptyset \wedge$$
$$|q_i.out| = |\{q_j | i < j \leq |T| \wedge \textit{(}q_i.out \cap q_j.in\textit{)} \neq \emptyset\}|$$

The algorithm described in Listing 3 refers to new concepts. These concepts need to be formally defined in order to be able to understand the process.

Definition 5.5.5. *(**Formal Concept Analysis**) Formal concept analysis (FCA) is a method of data analysis which describes a relationship between a particular set of objects and a particular set of attributes. FCA produces two kinds of output from the input data. The first is a concept lattice which is a collection of formal concepts in the data which are hierarchically ordered by a subconcept-superconcept relation. The second output of FCA is a collection of so-called attribute implications where an attribute implication describes a particular dependency which is valid in the data. Ganter et al. [22] provides the definition of FCA as a formal context $C = (O, A, R)$ where O is the set of objects, A is the set of attributes and $R \subseteq O \times A$ is the relation between objects and attributes. For the formal context, a formal concept c is defined as a pair (O_i, A_i) where $O_i \subseteq O$ and $A_i \subseteq A$ and every object in O_i has each attribute in A_i.*

In our case, objects O are SQL statements while attributes A are the clauses of a query as described in Section 5.3.2, where each concept c contains queries that implement the same data-manipulation functions, i.e. SQL statements of the same type and with equal clauses but with potentially different values taken by the input parameter. A concept groups the objects, i.e. queries, which have the same set of attributes.

Algorithm 3 Loop detection algorithm

```
Require: A set of concepts ($CL$) each containing a set of queries $c \in 2^T$ and the trace
    $T = \{q_1, ..., q_t\}$
Ensure: A set of potential loops $LS$
for all $C \in CL$ do
    for all $q \in C$ do
        $NBq = 0$
        $Nestedqueries = \{\}$
        $q^{-1} \leftarrow Prec(q)$
        if $(OutInDep(q^{-1}, q))$ then
            for all $r \in C$ do
                $r^{-1} \leftarrow Prec(r)$
                if $(r^{-1} = q^{-1} \wedge OutInDep(q^{-1}, r))$ then
                    $NBq++$
                    $Nestedqueries.add(r)$
                end if
            end for
            if $|QRes(q^{-1})| = NBq$ then
                $LS \leftarrow LS \cup (q^{-1}, Nestedqueries)$
            end if
        end if
    end for
end for
```

Cluster	**Queries**	**Clauses**

c_1	$\{q1, q6\}$	Selectclause =\{name, ncust\}
		Fromclause = \{customer\}
		Whereclause =\{(=,locality)\}

c_2	$\{q2, q3, q4, q5, q7, q8, q9\}$	Selectclause =\{norder, ncust, date\}
		Fromclause = \{order\}
		Whereclause =\{(=,ncust)\}

c_3	$\{q10\}$	Selectclause =\{(ncust, name), (ncust)\}
		Fromclause = \{customer, order\}
		Whereclause =\{(in, ncust)\}

Table 5.3: The clusters of the SQL queries of trace 5.4

Therefore, using the defined FCA, we first cluster queries belonging to the trace T and having the same set of clauses, possibly with different input values. Starting from this set of clusters, Algorithm 3 seeks to identify loops in the program based on the presence of O/I dependencies ($OutInDep : (2^T, 2^T) \rightarrow Boolean$) (the query results used as an input by other queries). By looking at the results obtained by a certain query and by checking the number of times its successive query occurs with different input values, it is possible to identify a possible loop in the program.

The algorithm starts by checking, for each query q of each concept C, whether there exists an O/I dependency between q and its nearest preceding query ($Prec(q)$) in the input trace belonging to a different cluster, i.e., query q^{-1}. If this condition is satisfied, the algorithm checks to see whether all or some of the other queries within the concept also have q^{-1} as a nearest preceding query in a O/I dependency (line 9). If this is the case, the algorithm checks to see if the query result ($QRes(q^{-1})$) retrieved by q^{-1} correspond to the number of queries in concept C (condition on line 14). This means that all or some of queries of cluster C have been successively executed once for each result of its preceding query. We can then make the hypothesis that there is a nesting relationship between the query q^{-1} and $Nestedqueries$.

Starting from the SQL trace of Figure 5.4, we generate 3 concepts (Table 5.3): C_1=$\{q1, q6\}$ contains a query having *name* and *ncust* as select clause, *customer* as from clause and *locality* as where clause; C_2=$\{q2$, $q3$, $q4$, $q5$, $q7$, $q8$, $q9\}$ regroups queries having *norder*, *ncust* and *date* as select clause, *orders* as from clause and *ncust* as where clause. Lastly, C_3=$\{q10\}$ contains

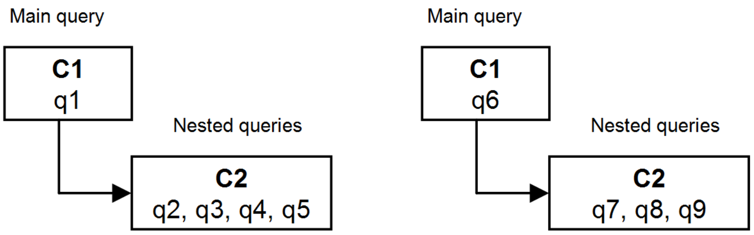


Figure 5.6: The loop detection results of the SQL queries of trace 5.4.

a query having *ncust* and *name* as select clause, *customer* as a from clause and a sub-query as a where clause.

When applying the algorithm to this set of concepts, we detect a nesting between the first query q_1 of concept C_1 and the queries of concept C_2=$\{q2$, $q3$, $q4$, $q5\}$ and between the query $q6$ of concept C_1 and the queries C_2=$\{q7, q8, q9\}$ (see Figure 5.6). Indeed, (i) the queries $\{q2$, $q3$, $q4$, $q5$ $\}$ in C_2 are preceded by $q1 = q2^{-1} = q3^{-1} = q4^{-1} = q5^{-1}$ (resp. $\{q7, q8, q9\}$ in C_2 are preceded by $q6$=$q7^{-1}$=$q8^{-1}$=$q9^{-1}$) (ii) it exists an O/I dependency between $q1$ and the queries $\{q2, q3, q4, q5\}$ in C_2 (resp. it exists an O/I dependency between $q6$ and the queries $\{q7, q8, q9\}$ in C_2) and (iii) the number of results of query $q1$ (resp. $q6$) is equal to the number of queries within concept C_2, i.e., ($|QRes(q1)|$=4 resp. $|QRes(q6)|$=3).

5.5.3 Synthesis

Analyzing dependencies between successive queries may help developers to understand how they depend on each other. However, it is worth noting that the loop detection algorithm and the O/I dependencies detection fail in some situations, such as:

- In the case where we do not have the SQL query results as input. Otherwise, we considered an alternative way to detect loops, where we rely on the logical schema of database, specifying the foreign keys between tables. First, we detect all couples of queries (qi, qj) where qi selects a primary key column of table Ti as an output and qj uses a foreign key column of table Tj which refers to the primary key of Ti as an input (or vice versa). Second, we gather all the couples of queries: ($q1$, $q2$), ($q1$, $q3$), ...,($q1$, qn) sharing the same $q1$ and $q2$, $q3$, ...qn having the same properties with possible different values. Then, we may assume that there is a loop between these queries, where $q1$ represents the main query and $q2$, $q3$, ...qn represent the nested queries.

- In the case where the program does not use the complete results set of the queries it executes. However, as part of the algorithm that we implemented, we considered only the case where the main query uses the complete set of its results by assuming that it is the common case.

5.6 Query Interpretation

The query interpretation phase requires two additional inputs: (1) the conceptual schema of the database, i.e., a platform-independent, more abstract specification of the application domain concepts, their attributes and relationships. This schema can be either available, or can be obtained via database reverse engineering techniques [27]. (2) the mapping between the conceptual and logical schemas, that specifies the correspondences between the logical schema objects (tables, columns, foreign keys) and the high-level concepts they implement (entity types, attributes, relationship types).

Using these two additional inputs, this phase automatically provides a conceptual interpretation of the SQL traces, expressed as a set of concept manipulation operations in order to answer the following research question: *What do these SQL traces mean in terms of domain-specific concept manipulation?*. To address this question, we divided this phase into three steps, namely (1) trace abstraction; (2) sub-schema annotation and (3) interpretation generation.

Main query	**Nested queries**	**Abstracted query**
Select $c1_1$, $c1_2$, ...$c1_n$ **from** T_1 **where** $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;	**select** $c2_1$,$c2_2$, ...$c2_n$ **from** T_2 **where** $col_2 = val_{c1_1}$ and $col2_1$ $op2_1$ $val2_1$ and... $col2_m$ $op2_m$ $val2_m$;	**select** $c1_2$, ...$c1_n$, $c2_1$,$c2_2$, ...$c2_n$ **from** T_1, T_2 **where** $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$ and $T_1.col_1 = T_2.col_2$ and $col2_1$ $op2_1$ $val2_1$ and... $col2_m$ $op2_m$ $val2_m$;
Select $c1_1$,$c1_2$, ...$c1_n$ **from** T_1 **where** $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;	**Delete from** T_2 **where** $col_2 = val_{c_1}$ and $col2_1$ $op2_1$ $val2_1$ and... $col2_m$ $op2_m$ $val2_m$;	**Delete from** T_2 **where** $col2_1$ $op2_1$ $val2_1$ and... $col2_m$ $op2_m$ $val2_m$ and $T_1.col_1 = T_2.col_2$ and $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;
Select $c1_1$,$c1_2$, ...$c1_n$ from T_1 where $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;	**Update** T_2 **Set** $col2_1 = exp2_1$, $col2_2 = exp2_2$,..., $col2_m = exp2_m$ **where** $col_2 = val_{c_1}$ and $col2_1$ $op2_1$ $val2_1$ and... $col2_m$ $op2_m$ $val2_m$;	**Update** T_2 **Set** $col2_1 = exp2_1$, $col2_2 = exp2_2$,..., $col2_m = exp2_m$ **where** $col2_1$ $op2_1$ $val2_1$ and... $col2_m$ $op2_m$ $val2_m$ and $T_1.col_1 = T_2.col_2$ and $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;
Select $c1_1$, $c1_2$, ...$c1_n$ from T_1 where $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;	**Insert into** T_2 ($c2_1$, $c2_2$, ..., $c2_n$) **Values** ($val2_1$, $val2_2$, ...,$val2_n$);	**Insert into** T_2 ($c2_1$, $c2_2$, ..., $c2_n$) **Select** $c1_1$, $c1_2$, ...$c1_n$ from T_1 where $col1_1$ $op1_1$ $val1_1$ and ... $col1_m$ $op1_m$ $val1_m$;

Table 5.4: Abstraction rules of nested queries

5.6.1 Trace Abstraction

This step depends on the previous level (inter-query analysis), where it requires the results of the dependencies detection and the loops detection. The main objective of this step is to abstract the SQL execution trace given as an input, more specifically the nested queries which we identified in the previous step (Loop detection algorithm 3). This means that the procedural join, implemented via successively nested queries, will be interpreted as one single join query. As result, we obtain a new SQL trace containing a set of abstracted queries representing the data manipulation behavior corresponding to the given program execution scenario.

To achieve this objective, we first defined a set of rules that represent the possible cases between one main query and a set of nested queries. Table 5.4 lists the required transformation for the four possible cases:

1. The *select/select* join query representing the case where the main query and nested queries are all *select* queries.
2. The *select/delete* join query representing the case where the main query is a *select* query and the nested queries are *delete* queries.
3. The *select/update* join query representing the case where the main query is a *select* query and the nested queries are *update* queries.
4. The *select/insert* join query representing the case where the main query is a *select* query and the nested queries are *insert* queries.

Algorithm 4 SQL trace abstraction algorithm

```
Require: $LS$ set of potential loops and the trace $T = \{q_1, ..., q_t\}$ containing
    a set of queries
Ensure: Abstracted trace $AT$ containing a set of abstracted queries
for all $q \in T$ do
    if $q \in MainQueryOf(LS)$ then
        $q_{abstracted} \leftarrow ApplyRulesNestedQueries(q)$
        $T \leftarrow T - (NestedQueries(q) - (NestedQueries(q) \cap Mainqueries(LS)))$
        $AT \leftarrow AT \cup q_{abstracted}$
    else
        $AT \leftarrow AT \cup q$
    end if
end for
```

Second, we defined an algorithm that implements these rules, which is described in Algorithm 4. The algorithm starts by checking, for each query q belonging to the trace T, if it represents a main query among the detected loops (Line 2) . If this is the case, the algorithm applies one of the corresponding rules in order to abstract it to an equivalent join query (Line 3). Then, the algorithm removes all the nested queries that do not play any role of the main query for other loops (Line 4).

Listing 5.4: The abstracted trace of trace 5.4

```
q'1: select customer.name, customer.ncust, order.norder, order.date from
    customer, order where customer.ncust = order.ncust and locality = 'Namur
    ';
q'2: select customer.name, customer.ncust, order.norder, order.date from
    customer, order where customer.ncust = order.ncust and locality = '
    Poitiers';
q'3: select customer.ncust, customer.name from customer where customer.ncust
     not in (select ncust from order);
```

By applying Algorithm 4 on the trace of Figure 5.4, we get a new SQL trace (Listing 5.4), where we abstract the two procedural joins ($q1$ with $q2, q3, q4, q5$ and $q6$ with $q7, q8, q9$) to two equivalent join queries $q'1$ and $q'2$.

This phase allows us (developers) to gain a better understanding of the program source code in terms of loops and to filter out from the input SQL trace all the repetitive queries in order to extract data manipulation functions followed by the program execution scenario at a higher level of abstraction.

5.6.2 Subschema Annotation

In order to generate the interpretation of each query, we need to manually annotate the resulting sub-conceptual schema by means of meta-properties (i.e., textual annotations that can be attached to schema constructs). These annotations will form the basis of the interpretation generation step. For instance, the attributes *Address* and *Locality* of the conceptual schema in Figure 5.2 will have *living in* as meta-property and the relationship type *place* will have two meta-properties. Each meta-property will describe the role of the relationship type according to the reading direction:

1. $Meta - property$ 1= "*who placed*", represents the left-hand-side role of the relationship type *place*. It will be used when the program looks for all those $customers$ *who placed* given $orders$.

2. $Meta - property$ 2= "*placed by*", represents the right-hand-side role of the relationship type *place*. It will be used when the program retrieves all $orders$ *placed by* some $customers$.

In the case where the conceptual subschema is not annotated, we treated the name of concepts and relationships as meta-properties.

5.6.3 Interpretation Generation

The third step of *the query interpretation* is *the interpretation generation*. This step requires as inputs: (1) the abstracted trace resulting from the first step and (2) the annotated sub-conceptual schema. Based on these inputs, this step provides an interpretation of each SQL query belonging to the abstracted trace. In the light of this, we chose two ways to represent this interpretation: (1) a natural language interpretation and (2) a visual interpretation.

5.6.3.1 Natural Language Interpretation:

Algorithm 5 Natural Language Interpretation algorithm

```
Require: The trace $AT = \{q_1, ..., q_t\}$ containing the abstracted queries and The conceptual schema $CS$
Ensure: Set of sentences $LS$
 1: for all $q \in AT$ do
 2:     $Concept \leftarrow ExtractAllConceptsOf(q, CS)$
 3:     $Level \leftarrow getLevelOfQuery(q)$
 4:     $FilteredConcept \leftarrow FilteringOutFrom(Concept, Level)$
 5:     $OrderedConcept \leftarrow OrderedAllConceptsOf(FilteredConcept)$
 6:     for all $c \in OrderedConcept$ do
 7:         if $ExistMetaProprietyOf(c, CS)$ then
 8:             $mp \leftarrow getMetaProprietyOf(c, CS)$
 9:             $sentence \leftarrow sentence \cup mp$
10:         else
11:             $sentence \leftarrow sentence \cup c$
12:         end if
13:     end for
14:     $LS \leftarrow LS \cup sentence$
15: end for
```

We define an algorithm (Algorithm 5) that seeks to provide a textual interpretation of an abstracted trace given as an input. It is expressed as a set of sentences in natural language. Based on the annotated conceptual subschema and the abstracted SQL trace resulting from the previous steps, the algorithm starts by extracting all concepts, attributes and relationships accessed by each query q belonging to the abstracted trace (line 1). According to the level number of the query (level 1 means that the query contains one sub-query), we filter out from the resulting set all unnecessary concepts for the interpretation (line 4). Then, the algorithm orders the subset of concepts (line 2). Finally, the algorithm generates the sentence corresponding to the current query by exploiting the conceptual schema annotations.

For instance, the concepts of the query $q'1$ (Listing 5.5) of the abstracted trace (Listing 5.4) are : $name, ncust, norder, date, customer, order, locality$. Then, they will be ordered as follows: $name \rightarrow ncust \rightarrow norder \rightarrow date \rightarrow order \rightarrow customer \rightarrow locality$. After, each concept is replaced by its meta-property, as follows: $name \rightarrow numberofcustomer \rightarrow numberoforder \rightarrow dateoforder \rightarrow order \rightarrow placedby \rightarrow customer \rightarrow living\ in$. Then, the algorithm generates the following interpretation: "*Retrieve name, number of customer, number of order and date of order placed by customer living in Namur*".

Listing 5.5: Query $q'1$ of the abstracted trace of the trace in Listing 5.4

```
q'1: select name, ncust, norder, date from customer, orders where customer
    .ncust = orders.ncust and locality = 'Namur';
....
```

5.6.3.2 Visual Interpretation

With this type of interpretation, we provide users with a visual interpretation of the abstracted SQL trace. To this end, we use the sub-conceptual schema as representation model. Starting from the clauses representing the parsed queries belonging to the abstracted trace, we classify them into two main categories. These are:

1. **The selection criteria:** This category includes the *where* and the *join* clauses. It attempts to graphically highlight the search criteria of the query.

2. **The output information:** This category contains the set of query results. According to the type of the query, this category includes a different clause: (1) in the case of a *select* query, this category includes the *select* and the *from* clauses; (2) in the case of a *delete* query, this category includes the *delete* clause; (3) in the case of a *insert* query, this category includes the *insert* clause and (4) in the case of a *update* query, this category includes the *update* and the *set* clauses.

In the next section, we will present a tool supported by the approach that implements this.

5.7 Tool Support

Here, we provide an overview of DAViS (**D**ynamic **A**nalysis and **Vi**sualization of the **S**QL execution traces) - a tool that allows us to support the defined approach, which helps us better understand the data-manipulation behavior of data-intensive systems via the visualization of the SQL execution traces analysis. The overall framework of the tool (see Figure 5.7) is implemented as a plug-in in an integrated tool, called DB-Main[2], a CASE tool supporting database design and evolution. The inputs required by DAViS are:

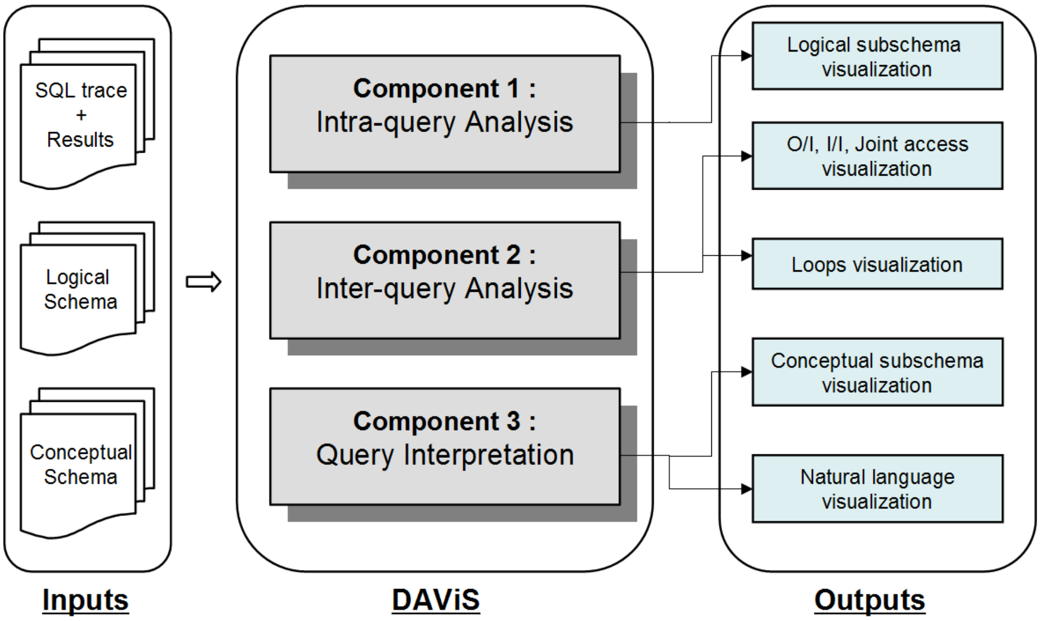

SQL trace
+
Results
Logical Schema
Conceptual Schema
Component 1 : Intra-query Analysis
Component 2 : Inter-query Analysis
Component 3 : Query Interpretation
Logical subschema visualization
O/I, I/I, Joint access visualization
Loops visualization
Conceptual subschema visualization
Natural language visualization
Inputs
DAViS
Outputs

Figure 5.7: Overall framework of DAViS

1. One SQL execution trace in the form of an XML file containing a set of SQL queries representing the execution program of the given scenario and its results;

2. The logical database schema using the basic Entity-Relationship (ER) model (e.g., tables, foreign keys, primary keys);

3. Its corresponding conceptual schema with the mapping between both (This third input is optional).

DAViS includes three components corresponding to each level (levels 1, 2 and 3) of the proposed approach, and it provides multiple visualizations including the analysis results of one or multiple components.

[2] http://www.db-main.be

5.7.1 Metrics

DAViS provides a way to visualize the analysis results of each level. It is a graph-based representation, where nodes represent an object (tables, columns, entities, attributes, etc.) and the edges represent a relationship between two objects. We opted for a graph-based representation in order to retain the same metaphors of logical schema and conceptual schema visualization proposed by DB-main. Furthermore, we consider the information visualized as the main contribution of this study. Thus, it is possible to choose other visual metaphors while keeping the same objective of the analysis.

Before presenting DAViS' interface, it is necessary to present the metaphors that it uses. Table 5.5 enumerates them in order to be able to understand the proposed visualizations. Here, we select a shape for each object and the size of the latter to represent its access frequency.

5.7.2 DB-MAIN

DB-MAIN [3] is a CASE tool (Computer-Aided Software Engineering) supporting database design and evolution. It is designed to help developers and analysts in most data engineering processes, such as design processes (conceptual design, logical design, etc.), transformations (schema transformation, model transformation, etc.), reverse engineering (schema analysis, code analysis, etc.), maintenance and evolution (database evolution, impact analysis, etc.).

Moreover, DB-MAIN includes meta-modeling components that allow one to develop new functions and to extend its repository through plug-ins. Besides this, DB-MAIN was chosen to represent the database schemas. DAViS is implemented as a plug-in.

5.7.3 JUNG Library

In order to implement DAViS, we used JUNG API [4] (**J**ava **U**niversal **N**etwork/ **G**raph Framework) a JAVA software library specially designed to model, analyze and visualize data that can be represented by a graph or network. We opted for this library due to the fact that it supports a wide variety of entity representations and their relations, such as directed and undirected graphs and graphs with parallel edges. It also allows us to annotate graphs, entities, and relations with meta-data. Here, we used it to implement a graph-based representation using our own metaphors.

[3] http://www.rever.eu/fr/db-main

[4] http://jung.sourceforge.net/

	Element	**Metaphor**
Nodes	- Table - Entity	- A rectangle containing the name of the table/entity
	- Column - Attribute	- An oval containing the name of the column/attribute
Edges	- Relationship (conceptual visualization)	- An hexagon containing the name of the relationship
	- I/I dependency - O/I dependency - Join query	- A dotted line between two columns
	- Join access	- A dotted line between two tables
	- Foreign key	- An arrow between two tables
	- Membership	- A line between entity-attribute, table-column, entity-relationship
Others	- Element frequency	- The size of the object - The thickness of the link

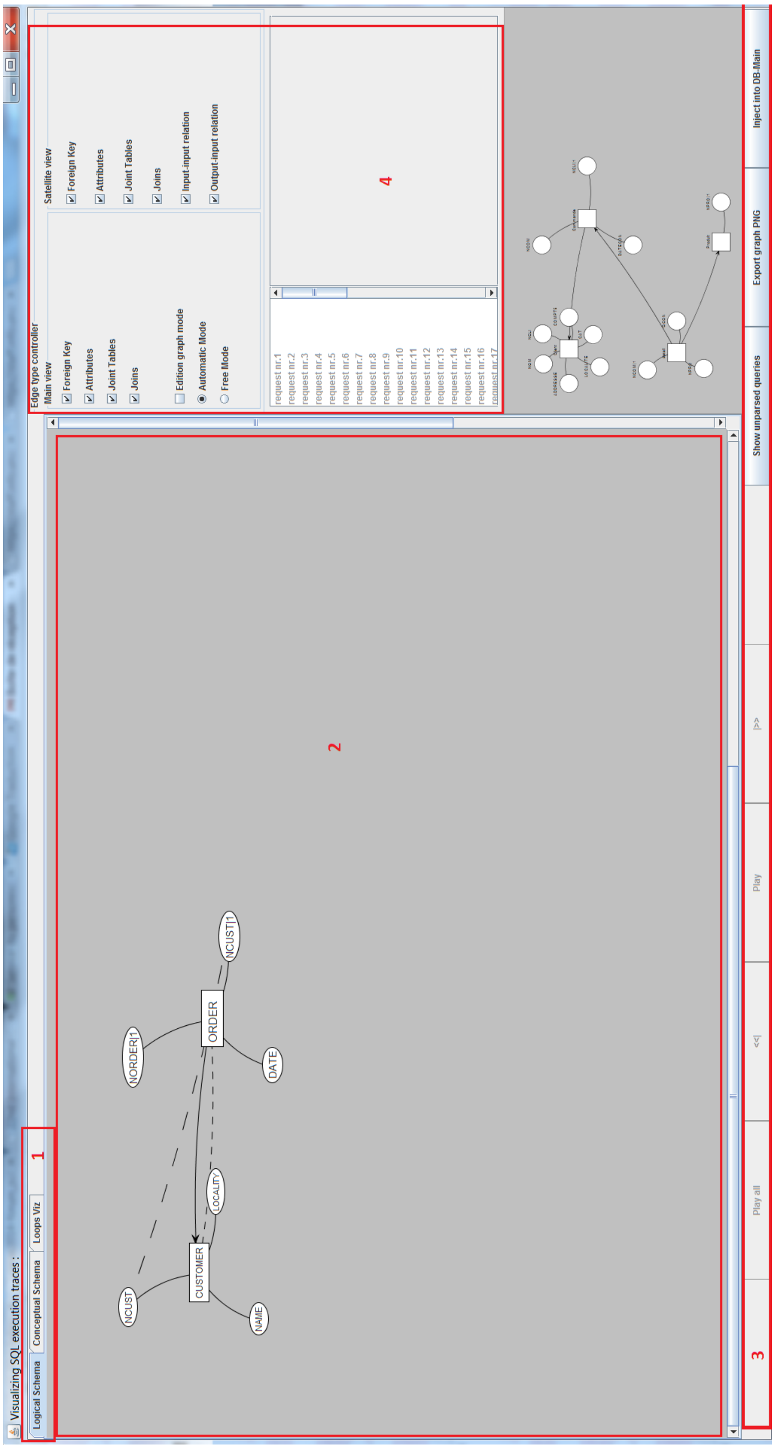
Table 5.5: Visual metaphors used by DAViS

5.7.4 DAViS User Interface

Figure 5.8 depicts the interface of DAViS, which includes five important panels:

1. **Components panel:** This panel includes all the proposed visualization by DAViS (logical visualization, loops visualization and conceptual visualization). It allows users to switch from one visualization to another.

2. **Main view panel:** As its name suggests, this panel seeks to display all the proposed visualizations, such as: sub-logical schema, sub-conceptual schema, dependencies and loops.



Edge type controller

Main view

☑ Foreign Key

☑ Attributes

☑ Joint Tables

☑ Joins

☐ Edition graph mode

● Automatic Mode

○ Free Mode

Satellite view

☑ Foreign Key

☑ Attributes

☑ Joint Tables

☑ Joins

☑ Input-input relation

☑ Output-input relation

Figure 5.8: DAViS user interface

3. **Graph evolution panel:** This panel contains a set of features that allows us to dynamically manipulate the chosen visualization. For instance, it might play the evolution of the subschema affected by the trace. It also allows one to export the current visualization into a ”png” format and/or inject the information contained in the visualization as meta-properties in DB-main.

4. **Control panel:** This panel contains a set of features that allows users to choose the information that they want to display, by filtering out from the current visualization any unnecessary information. For instance, it may just display only one type of dependency or only display only tables without their columns.

5.7.5 Visualization Modes

In addition to the visual metaphors, DAViS incorporates two visualization modes depending on the needs of users: (1) *the global visualization*, which involves visualizing the global impact of the whole SQL statements belonging to the trace given as an input on database; (2) *the local visualization*, where it just focuses only on one SQL statement in order to understand its meaning.

5.7.5.1 Global Visualization

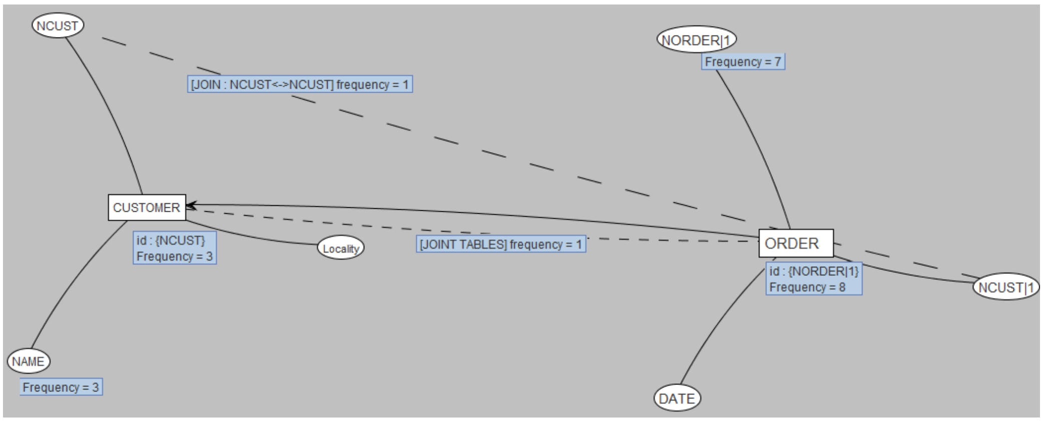


Figure 5.9: The global visualization of the trace of Figure 5.4

It worth noting that the input trace is given in an XML file containing a set of SQL statements executed in the order of appearance in the latter. From this context, we thought of implementing a global visualization, where it allows us on the one hand to visualize statically the final state as well as the intermediary states of the database portion affected by the trace given as an input; on the other hand, it allows us to visualize dynamically the evolution of the database portion affected by the trace from the first SQL statement to the last SQL statement belonging to the trace. Figure 5.9 shows the global visualization of the SQL execution trace of the example described in Listing 5.4.

Through this visualization mode, we can extract useful information that will help users to gain an overall understanding of the data-manipulation behavior of the given trace. This information can be grouped into two categories, these being:

- **Sensitive objects:** We consider having a global visualization of the objects impacted in the trace given as an input with their access frequencies, which allows users to identify not only objects to focus on, but also the degree of their importance. Such information can be seen as a support for the detection of some related program problems. For instance, the problem generated by the excessive use of only one table (entity) in one execution scenario. In such a case, this could be an indicator of a database design problem. Another example is the case of the called *God table*, which contains hundreds of (unrelated) columns, and it used by almost all programs.

- **Related objects:** For most legacy databases, some integrity constraints may be implicit, which means they exist but they have not been explicitly declared in the DDL code. This is because most legacy database management systems do not allow the explicit declaration of foreign keys (e.g. MySQL-MyISAM) and therefore, when we want to reverse engineer the database, the recovery of these constraints is necessary, which is not an easy task. In this context, the global visualization of the inter-query analysis (where we detect all implicit dependencies between queries) facilitates the recovery of such relationships. For instance, from the joint access dependency (the tables that are accessed jointly) and especially this with a high access frequency, we can assume that there is an undeclared *Foreign Key* between these tables.

5.7.5.2 Local Visualization

In contrast to the global visualization, which seeks to understand the global impact of the given trace on the database of the studied system, the local visualization seeks to provide a support to understand the meaning of each SQL statement belonging to the given trace. We will focus on the behavior of a single SQL statement in order to understand its meaning and consequently understand its corresponding source code. The proposed visualization is dynamically animated and it has two phases. These are:

1. ***Input visualization:*** This phase concerns the visualization of the selection criteria of the SQL query if it exists.

2. ***Output visualization:*** This phase concerns the visualization of the output part of the SQL query, which consists of the query results, such as select clause, update clause and insert clause.

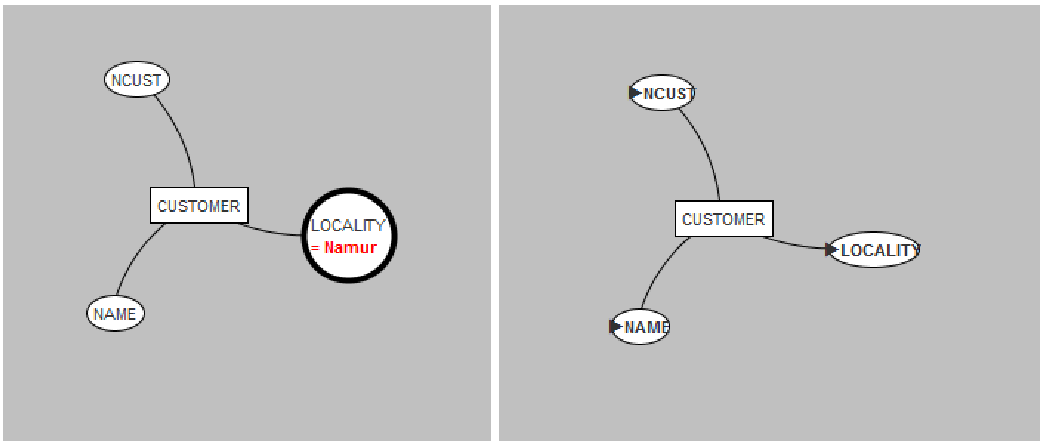


Figure 5.10: The local visualization of the first SQL statement of the SQL trace 5.4 (Left: the input, Right: the output visualization)

Figure 5.10 shows the local visualization of the first SQL statement belonging to the trace of the example described in trace 5.4, where the query attempts to select the *name, ncust, norder and date* of orders placed by customers living in *"Namur"*. The left hand figure shows the first phase, which is the selection condition, while the right hand figure shows the result of the query which *name, ncust, norder and date* of the customer.

5.7.6 Component 1: Intra-query Analysis Visualization

The first component of DAViS is the *intra-query analysis*, based on the logical schema, this component highlighting the database *subschema* involved in the given trace, i.e., the set of tables and columns involved in the SQL execution trace given as input. The proposed visualization leads to a technical understanding of the given trace, in terms of a global impact on the database.

It worth mentioning that the minimal inputs of this component are a single SQL execution trace and the logical schema of the system's database. Otherwise, the conceptual schema is provided with the mapping between both (logical and conceptual) schemas. This component also allows one to visualize the conceptual subschema affected by the trace using the same filtering algorithm.

In addition, each object belonging to the visualized subschema is provided with its access frequency, which is visually represented by the size of this object. This helps users to understand what the most sensitive objects are for possible modification.

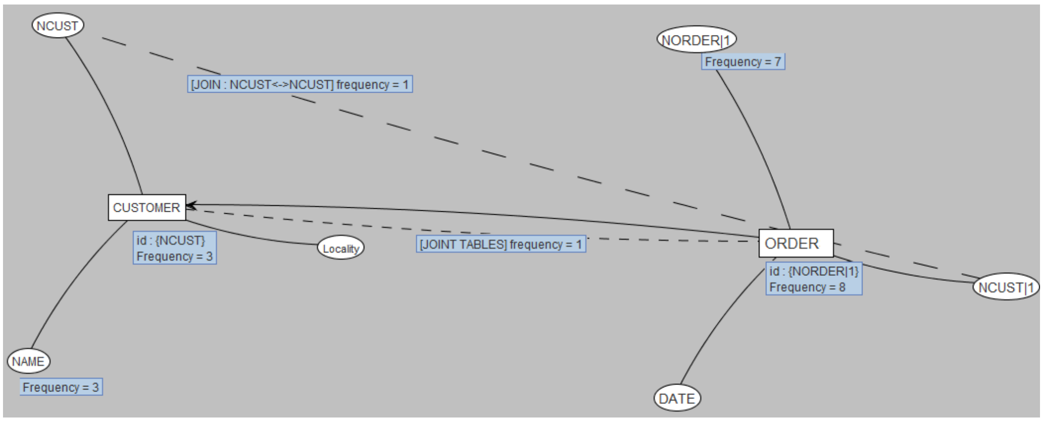
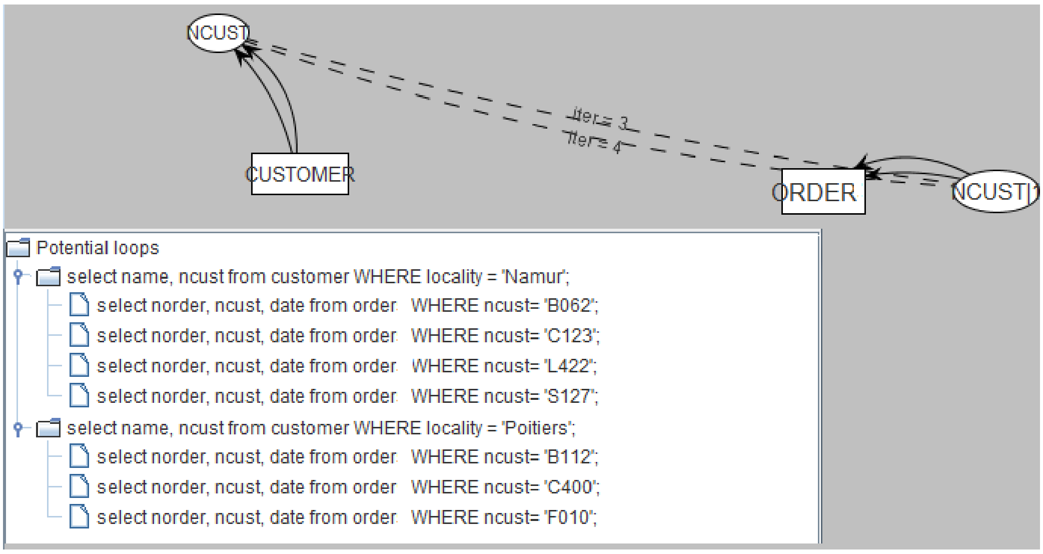


Figure 5.11: The sub-schema affected by the SQL execution trace of the trace of Figure 5.4

Figure 5.11 shows the logical subschema of trace 5.4. From the visualization, we can see that all the objects present are used by the trace with a different frequency. More precisely, the trace uses 2 of 4 initial tables, where table *order* and columns *norder* and *ncust* are the most frequently used with an access frequency equal to *7*.

5.7.7 Component 2: Inter-query Analysis Visualization

The second component of DAViS is the *inter-query analysis*, based on the logical schema of the program database, this component visually emphasizing implicit dependencies between successive SQL queries involved in the captured SQL execution trace. This component leads to a technical understanding of how queries depend on each other by visualizing the detected dependencies. These dependencies may be grouped into two categories, these being: (1)



Potential loops
- select name, ncust from customer WHERE locality = 'Namur';
  - select norder, ncust, date from order WHERE ncust= 'B062';
  - select norder, ncust, date from order WHERE ncust= 'C123';
  - select norder, ncust, date from order WHERE ncust= 'L422';
  - select norder, ncust, date from order WHERE ncust= 'S127';
- select name, ncust from customer WHERE locality = 'Poitiers';
  - select norder, ncust, date from order WHERE ncust= 'B112';
  - select norder, ncust, date from order WHERE ncust= 'C400';
  - select norder, ncust, date from order WHERE ncust= 'F010';

Figure 5.12: Loop visualization

one-to-one dependency: This category includes all dependencies which link two queries, such as *output/input* dependency (O/I), *input/input* dependency (I/I), joint access dependency, foreign-key and membership; (2) **one-to-many dependency:** This category includes the dependency between a set of queries, and it mainly focuses on the nested queries.

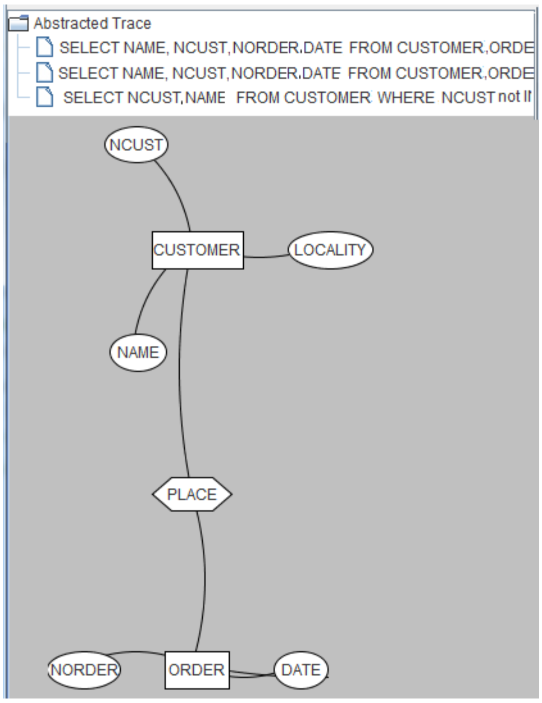
According to these two categories, the inter-query analysis component provides two visualizations. The first one is the visualization of all one-to-one dependencies based on the logical schema representation, while the second one is the visualization of all nested queries. The component also proposes tree-based visualization for the one-to-many dependency, where we can get the details of each detected nested query.

Figure 5.12 shows the loop visualization of the detected loops from the trace of Figure 5.4. In the sample trace given as input, there are two loops between *order* and *customer* on column *ncust* with 4 and 3 iterations, respectively. We can also run through the tree-based representation on the right of Figure 5.12, where we can inspect separately each of the nested queries.

5.7.8 Component 3: Query Interpretation Visualization

The third component of DAViS is the query interpretation component, which provides a visualization for the abstracted trace using a tree-based visualization as a representation model. Figure 5.13 shows the abstracted trace obtained from the SQL trace of Listing 5.4 given as input. In addition, it provides two interpretation modes (visual interpretation and natural language interpretation).

5.7.8.1 Conceptual Interpretation



Abstracted Trace

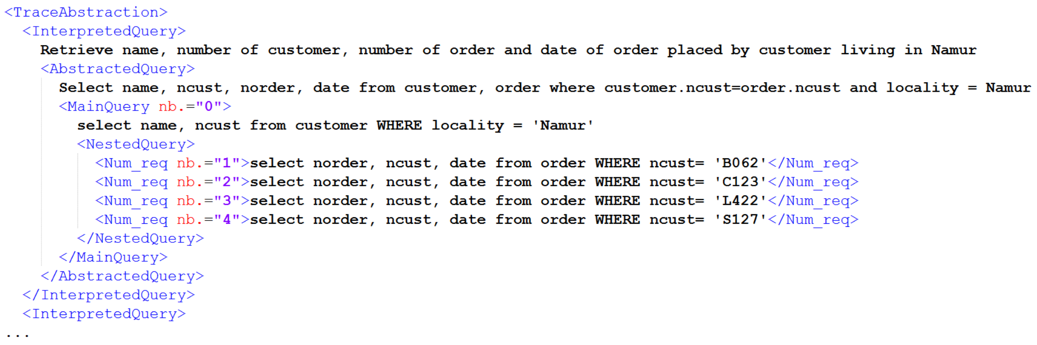
```
SELECT NAME, NCUST,NORDER.DATE  FROM CUSTOMER,ORDE
SELECT NAME, NCUST,NORDER.DATE  FROM CUSTOMER,ORDE
SELECT NCUST,NAME   FROM CUSTOMER WHERE NCUST not I
```

Figure 5.13: Trace abstraction and visual interpretation

Based on the conceptual schema representation, this component provides a visualization of the conceptual sub-schema affected by the SQL trace. Here, it allows us to visualize the effect of each query q belonging to the abstracted trace on the conceptual schema in order to understand the meaning of the query from an abstract perspective. Figure 5.13 shows the abstracted trace and its corresponding conceptual subschema, where we can select any query from the abstracted trace in order to visualize its effect dynamically.

5.7.8.2 Natural Language Interpretation

The natural language interpretation generates an external *xml* file containing a set of sentences expressed in natural language, as shown in Figure 5.14. This interpretation helps users to understand data-manipulation functions from an abstract point of view.



```
<TraceAbstraction>
  <InterpretedQuery>
    Retrieve name, number of customer, number of order and date of order placed by customer living in Namur
    <AbstractedQuery>
      Select name, ncust, norder, date from customer, order where customer.ncust=order.ncust and locality = Namur
      <MainQuery nb.="0">
        select name, ncust from customer WHERE locality = 'Namur'
        <NestedQuery>
          <Num_req nb.="1">select norder, ncust, date from order WHERE ncust= 'B062'</Num_req>
          <Num_req nb.="2">select norder, ncust, date from order WHERE ncust= 'C123'</Num_req>
          <Num_req nb.="3">select norder, ncust, date from order WHERE ncust= 'L422'</Num_req>
          <Num_req nb.="4">select norder, ncust, date from order WHERE ncust= 'S127'</Num_req>
        </NestedQuery>
      </MainQuery>
    </AbstractedQuery>
  </InterpretedQuery>
  <InterpretedQuery>
...
```

Figure 5.14: Natural language interpretation

5.8 Case Studies

In order to evaluate the approach defined in this chapter as well as DAViS (the tool supported by the analysis), this section presents two case studies in which we evaluate DAViS to show how it facilitates the understanding of SQL execution traces. To achieve this goal, we have selected two open-source systems. These two systems were chosen for the reasons that they provide different features, each accessing a different portion of an underlying database to support different activities. They can be considered good representative for data-intensive systems i.e., systems where most of the complexity is concealed in their interactions with their database. Even though these systems are of limited size and complexity, they demonstrated sufficiently the capability of the proposed approach in real environments. With this in mind, we also mitigated the quality of the input traces by analyzing features of a different nature. These are:

1. ***WebCampus:*** WebCampus is an open-source Learning Management Systems (LMS) that allows teachers to manage learning and collaboration activities on the Web and to offer on-line courses to their students. This system consists of almost half a million of lines code written in PHP and it exploits a MySql database consisting of 33 tables and 198 columns, representing the data on available on-line courses, course users, university departments, etc. Figure 5.15 shows the logical schema describing the application domain of WebCampus.

2. ***WebDeb:*** WebDeb is a collaborative open-source platform called *WebDeb*[5], designed to capture a debate's arguments with their actors, define relations between them, and run a Natural Language Processing

[5] https://www.webdeb.be/

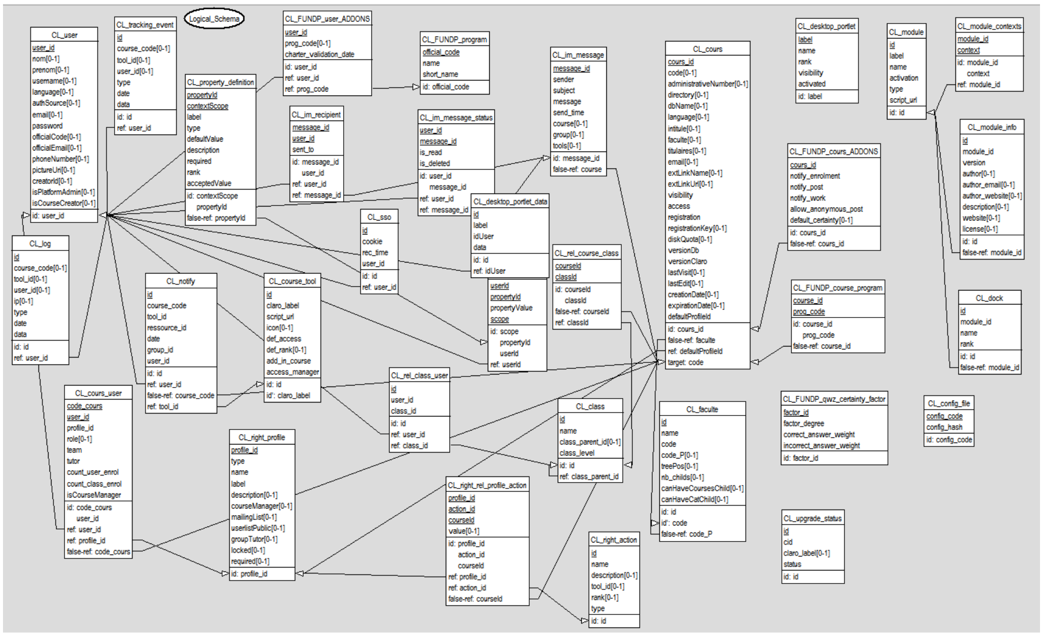


Figure 5.15: Logical schema of WebCampus application

analysis. More specifically, it is a Web application that permits the logical storage of arguments used in any kind of public debate (political, scientific, social, etc.). It is based on the fact that any text is made of a complex blend of assertions, which may take the form of a descriptive, prescriptive, appreciative or performative statement, and they are treated as «basic blocks». *WebDeb* allows one to display the various positions and justifications of different actors on each assertion and, also, to map the links between assertions. Figure 5.16 shows the logical schema describing the application domain of WebDeb.

System	**Language**	**# Table**	**# Column**
WebCampus	PHP/MySql	33	198
WebDeb	Java/MySql	55	231

Table 5.6: The systems description

Table 5.6 provides some statistics about the two applications in question. Here the first system's database (WebCampus) contains 33 tables (e.g. CL_cours, CL_notify, CL_module, etc.), each with its respective set of columns. The second system's database (WebDeb) is structured according

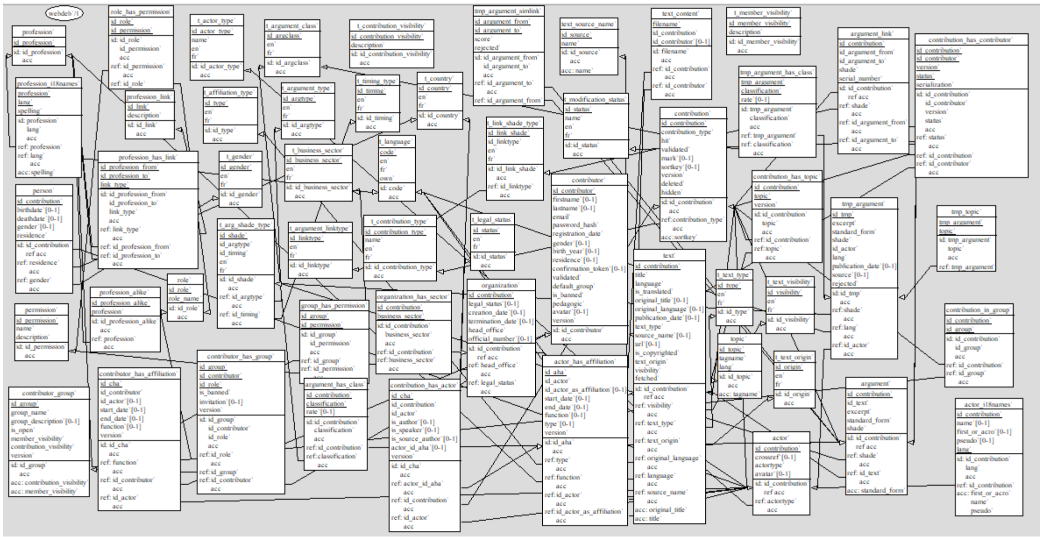


Figure 5.16: Logical schema of WebDeb application

to the various objects (statements, actors, references, themes, etc.), which are connected between them. It is a MySql database consisting of 55 tables and around 231 columns.

5.8.1 Traces Capturing

5.8.1.1 WebCampus

Examining the support of the WebCampus administrator *Jean Roch Meurisse*[6], who provided us a collection of SQL execution traces, each trace contains a set of SQL statements belonging to one of the 14 different scenarios, namely: *install course tool, add course manager, add course user, create course, delete course, delete course user, delete faculty attempt, install applet, uninstall applet, uninstall course tool, user register to course, user register to webcampus, user unregister from course,* and *register user.* Tables 5.7 lists the characteristics in terms of the size of the traces (Statements), the size of the trace results (Results) and the type of queries (Types) by scenario.

5.8.1.2 WebDeb

In order to be able to collect SQL traces of different scenarios, we locally installed WebDeb. Then, using JDBC logger, we collected the SQL execution traces corresponding to six typical interaction scenarios, namely: *registration,*

[6] https://directory.unamur.be/staff/jmeuriss

Scenario	**Query**	**Select**	**Insert**	**Delete**	**Update**	**Results**
install course tool	2169	2039	126	4	0	24180
add course manager	194	190	4	0	0	2391
add course user	155	151	4	0	0	1908
create course	29	20	9	0	0	299
delete course	132	124	1	7	0	1700
delete course user	84	83	0	1	0	996
delete faculty	37	37	0	0	0	419
install applet	88	82	4	0	2	721
uninstall applet	78	68	0	9	1	573
uninstall tool	1896	1888	0	8	0	22419
register to course	64	63	1	0	0	708
register to WebCampus	32	30	2	0	0	184
unregister from course	19	17	1	1	0	155
register user	27	24	3	0	0	163
Total	**5004**	**4816**	**155**	**30**	**3**	**56816**

Table 5.7: Characteristics of the parsed queries of the 14 scenarios of WebCampus application

add new actor, add new actor2, delete an actor, search an actor, validate affirmation. Table 5.8 describes the content of each scenario in terms of the number of statements, size of the traces results, and the type of queries.

5.8.2 Queries Parsing

We started by evaluating the developed parser on both systems. Here, Table 5.9 summarizes the parser results for both systems:

1. **WebCampus:** starting from 10,187 queries distributed on 14 scenarios, the parser filtered out 5,026 queries which were out of the logical schema and it was not able to parse 157 queries. As result, we obtained 5,004 parsed queries that represent 96.95 % of the initial queries.

2. **WebDeb:** starting from 4128 initial queries distributed on 6 scenarios, the parser filtered out 820 queries that were out from the given logical schema and it was not able to parse 409 queries. As result, we obtained 2899 parsed queries (2877 select, 11 insert, 4 delete and 7 update) that represent 91.94% of the initial queries.

Scenario	**Query**	**Select**	**Insert**	**Delete**	**Update**
registration	270	266	2	0	2
add new actor	855	851	1	1	2
add new actor 2	353	345	6	1	1
delete an actor	263	260	1	1	1
search an actor	905	902	1	1	1
validate affirmation	253	253	0	0	0
Total	**2899**	**2877**	**11**	**4**	**7**

Table 5.8: Characteristics of the parsed queries of the 6 scenarios of WebDeb application

# of queries	**WebCampus**	**WebDeb**
Initial queries	10.187	4128
Out of the schema	5026	820
Unparsed queries	157	409
Parsed queries	5004	2899
Select query	4816	2877
Insert query	155	11
Delete query	30	4
Update query	3	7

Table 5.9: The parser's results

It worth noting that most of the unparsed queries are due to the presence of SQL functions which the parser cannot parse (e.g. FOUND-ROWS()). This is due to the SQL grammar used by the « JSqlParser » library that we used.

5.8.3 Sub-schemas Results

Table 5.10 (resp. Table 5.11) provide statistics about the subschemas' size (tables and columns) affected by each scenario. From the results, we observe that we reduce on average 2/3 of objects from the initial logical schema of WebCampus and 1/2 objects from the initial logical schema of WebDeb, which leads us to suppose that the understanding of the subschema affected by each trace may reduce the cognitive effort than if we would have had the whole schema.

Figure 5.17 supports our observations, where the left hand figure shows

Scenario	# of tables	% of tables	# of columns	% of columns
install applet	7/33	21.21	32/198	16.16
add course manager	11/33	33.33	77/198	38.88
add course user	11/33	33.33	77/198	38.88
create course	11/33	33.33	62/198	31.31
delete course	12/33	36.36	82/198	41.41
delete course user	11/33	33.33	69/198	34.84
delete department	6/33	18.18	26/198	13.13
install tool	10/33	30.30	65/198	32.82
register user	9/33	27.27	55/198	27.77
uninstall tool	8/33	24.24	49/198	24.74
register to a course	9/33	27.27	59/198	29.79
register to WebCampus	9/33	27.27	56/198	28.28
unregister from course	9/33	27.27	58/198	29.29
uninstall tool	6/33	18.18	30/198	15.15

Table 5.10: Statistics about the Webcampus subschemas analysis

Scenario	# of tables	% of tables	# of columns	% of columns
registration	24/55	43.63	96/231	41.55
add new actor	24/55	43.63	96/231	41.55
add new actor2	21/55	38.18	88/231	38.09
delete an actor	17/55	30.90	73/231	31.60
search an actor	24/55	43.63	103/231	44.58
validate affirmation	13/55	23.63	57/231	24.67
search an actor	24/55	43.63	103/231	44.58

Table 5.11: Statistics about the WebDeb subschemas analysis

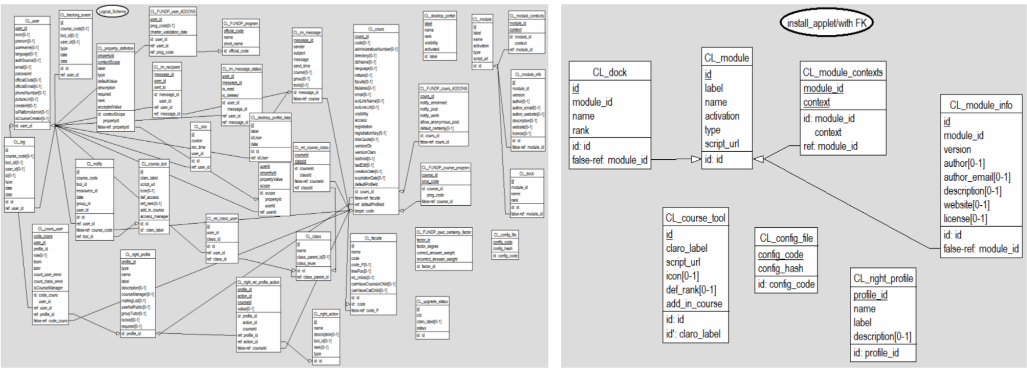


Figure 5.17: Left: The logical schema of WebCampus, right: The subschema of the install_applet scenario

the initial logical schema of WebCampus and the right hand figure depicts the sub-schema impacted by the *install_applet* scenario. The same applies to Figure 5.18, where the left hand figure shows the initial logical schema of WebDeb and the right hand figure shows the sub-schema affected by the *validate affirmation* scenario.

In addition to the fact that the subschema visualization provided by DAViS seeks to highlight only objects that are necessary to be analyzed to better understand the data-manipulation behavior of the SQL execution

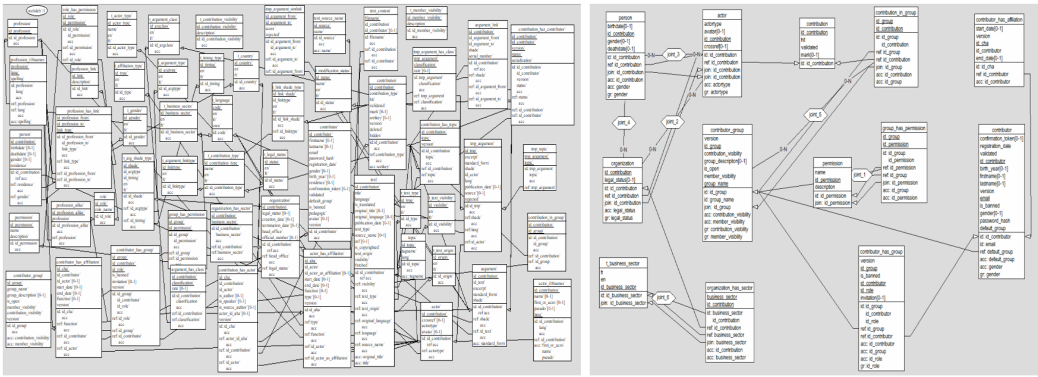


Figure 5.18: Left: The logical schema of WebDeb, right: The subschema of the validate affirmation scenario

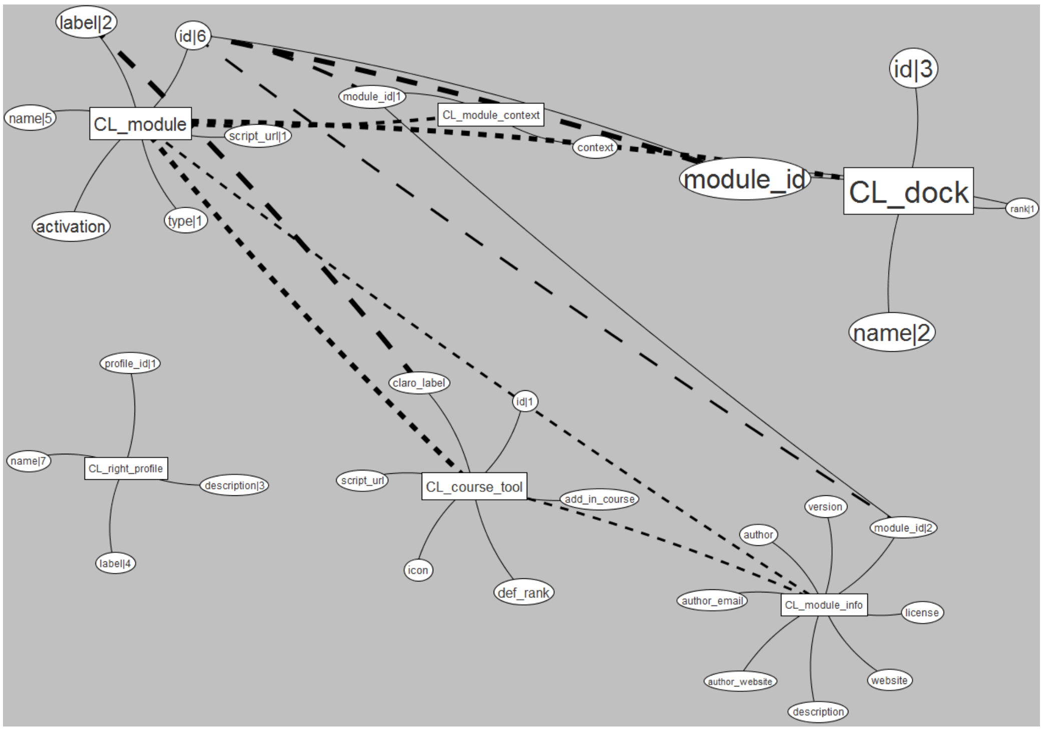


Figure 5.19: The detected dependencies of the *install_applet* senario of WebCampus

trace given as an input, DAViS provides the access frequency of each object belonging to this subschema:

- Tables 5.12 summarizes the access frequencies of all tables belonging to the *install_applet* logical subschema. From the results, we can see that

Tables	**Access frequency**
CL_module	23
CL_module_contexte	4
CL_course_tool	14
CL_dock	**53**
CL_moduel_info	3
CL_right_profile	4
CL_config_file	1

Table 5.12: The access frequencies of the tables of the scenario install_applet

Tables	**Access frequency**
Person	12
Actor	8
Contribution	4
Contribution_in_group	4
Contributor_has_affiliation	32
Organization	6
Contributor_group	36
Permission	34
Group_has_permission	34
Contributor	**64**
Contributor_has_group	**64**
Organization_has_sector	1
T_business_sector	1

Table 5.13: The access frequencies of the tables of the scenario validate affirmation

there are three key tables (CL_dock, CL_module and CL_course_tool), one of which has as frequency of 53. Here, the latter must be taken into account in the case of any changes.

- Tables 5.13 summarizes the access frequencies of all tables belonging to the *validate affirmation* logical subschema. From the results, we can see that there are six key tables (Contributor_has_affiliation, Contributor_group, Permission, Group_has_permission, Contributor, Contributor_has_group), two of which have a frequency of 64.

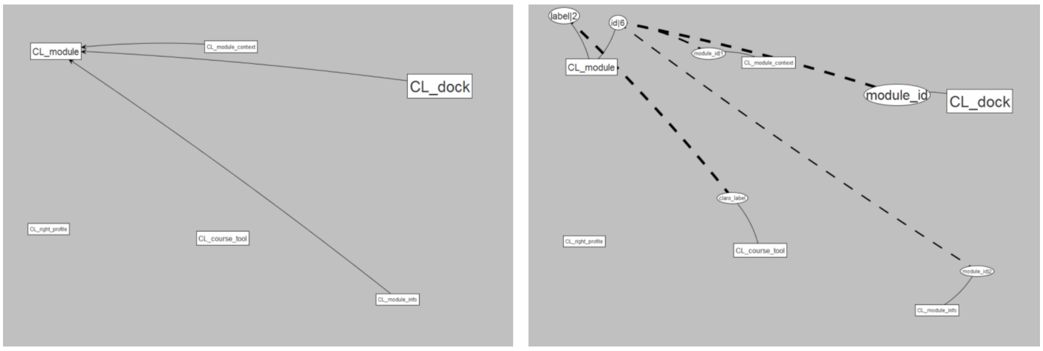


Figure 5.20: Left: The foreign key dependency of the *Install_applet* scenario, right: The join query dependency of the *Install_applet* scenario

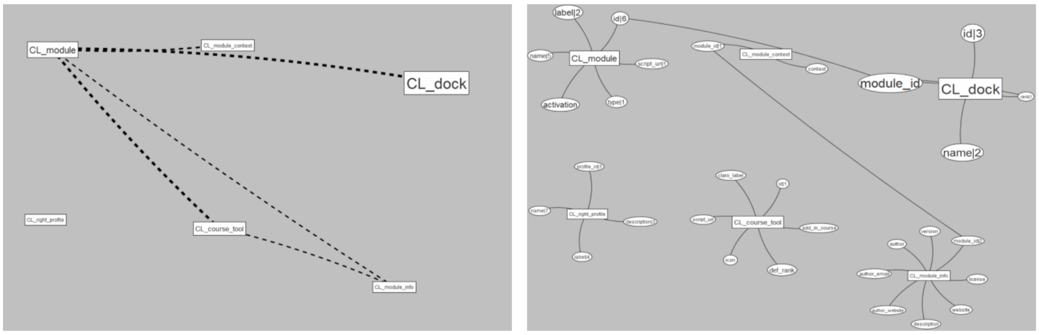
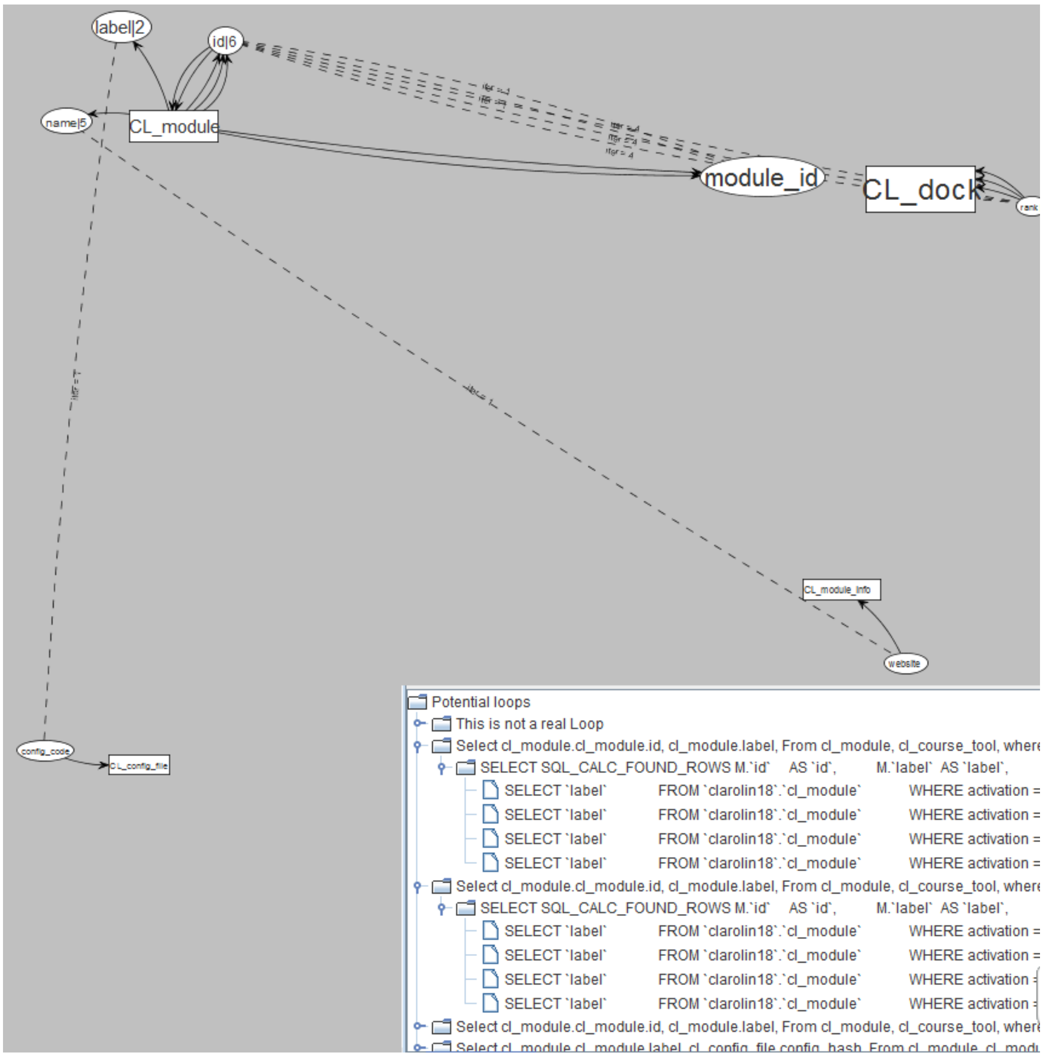


Figure 5.21: Left: The input-input dependency of the *Install_applet* scenario, right: The joint access dependency of the *Install_applet* scenario

5.8.4 Dependency Results

After extracting the subschemas affected by the different scenarios of the two systems, we were interested in the dependencies existing between queries of the same scenario (inter-queries analysis). Figure 5.19 shows the visualization of the detected dependencies between queries belonging to the *install_applet* scenario. Here, each kind of dependency is represented by a different visual representation. Figures 5.20 and 5.21 show each of them (foreign keys, join query, joint access and input-input dependencies). It should be added that each dependency is provided by the number of times it is used (frequency), which is visually represented by the thickness of the link.

Figure 5.22 shows the nested queries detection of the same scenario *Install_applet* of WebCampus. From the figure, we can see that there are two kinds of visualization, namely (1) graph-based visualization: where we have the number of loops (we detected 6 loops), for each detected loop, the num-



```
Potential loops
  This is not a real Loop
  Select cl_module.cl_module.id, cl_module.label, From cl_module, cl_course_tool, where
    SELECT SQL_CALC_FOUND_ROWS M.`id`    AS `id`,        M.`label`  AS `label`,
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
  Select cl_module.cl_module.id, cl_module.label, From cl_module, cl_course_tool, where
    SELECT SQL_CALC_FOUND_ROWS M.`id`    AS `id`,        M.`label`  AS `label`,
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
      SELECT `label`          FROM `clarolin18`.`cl_module`          WHERE activation =
  Select cl_module.cl_module.id, cl_module.label, From cl_module, cl_course_tool, where
  Select cl_module.cl_module.label, cl_config_file.config_hash, From cl_module, cl_modu
```

Figure 5.22: The detected loops of the *install_applet* scenario of WebCampus

ber of the nested queries; (2) tree-based visualization: where we have access to the details of each detected loop via a tree (abstracted query, main query and nested queries).

5.9 Conclusions

In this chapter, we presented an approach to assist the understanding of data manipulation behavior from the analysis of the SQL execution traces of data-intensive systems using visualization and dynamic analysis. We divided the approach into three major levels. These were: (1) the intra-query analysis, where we highlighted the subschemas affected by the input SQL trace; (2) the inter-query analysis, where we extracted all implicit dependencies between successive queries and (3) the query interpretation, where we interpreted the meaning of the trace in terms of domain-specific concept manipulation based on conceptual schema. Afterwards, we presented DAViS - a tool-supported approach, and we demonstrated its correctness and usefulness in two real case studies.

In the next chapter, we will present an approach that seeks to support the extraction of data-manipulation processes of data-intensive systems via an analysis of multiple SQL execution traces.

Chapter 6

Extracting Data Manipulation Processes from SQL Execution Traces

Strength does not come from physical capacity.
It comes from an indomitable will.
Mahatma Gandhi

Contents

6.1 Introduction 105
6.2 Illustrative Scenario 106
6.3 Approach and Research Questions 108
6.4 Data-oriented Properties 110
6.5 Process Mining 117
6.6 Evaluation 121
6.7 Conclusions 127

6.1 Introduction

As we said earlier, data-intensive systems are mainly characterized by a database and a set of applications performing frequent and continuous interactions with the former. Maintaining and evolving such systems can be effectively performed only after the system behavior has been sufficiently understood. In the previous chapter, we analyzed a single SQL execution trace so as to be able to extract the data-manipulation behavior of data-intensive systems in terms of database usage, accesses and dependencies. However, program understanding is not just about understanding the program's access to its database, but also about understanding their data-manipulation processes. To this end, in this chapter, we are going to analyze *multiple* SQL execution traces in order to recover the missing programs' processes starting from their logic of interactions with the database.

This chapter is structured as follows. We start by presenting a motivating example for our approach to data-manipulation process recovery. Then, we present our approach as well as the related research questions. Lastly, we present a validation of our approach and we discuss the experimental results before drawing some conclusions.

This study was carried out in collaboration with *Marco Mori* as part of his postdoctoral project. The results were published in conference proceedings [54, 53].

6.2 Illustrative Scenario

We consider an e-commerce Web store, where store administrators and customers access the system in order to sell and to buy products respectively in a worldwide area. In addition to the activities performed by stakeholders, administrators have to maintain available products with attributes and categories, check the current stock of products, issue orders to manufactures, manage basics setting of the store, manage payments and shipping methods, manage tax zones with tax rates, manage banners, newsletters and reviews. And performing these activities, stakeholders may query a big database by means of SQL-statements each belonging to one of the possible features, namely: *view products*, *user registration*, *payment registration*, *issue order*, *register manufacturer*, *manage reviews*, *manage banners* and *manage newsletters*.

Each feature requires frequent and continuous interactions with the database to fulfill its objective. For instance, the *view products* feature has to access the information about categories, products in the store, detailed information about selected products such as product price, product characteristics and manufacturer information. What is more, since products can be searched based on different parameters, we may have different accesses to the database based on the selection criteria, i.e., in the case of a category-driven search, the application accesses the corresponding category information, while in the case of manufacturer-driven searches, the application will alternatively access manufacturer information.

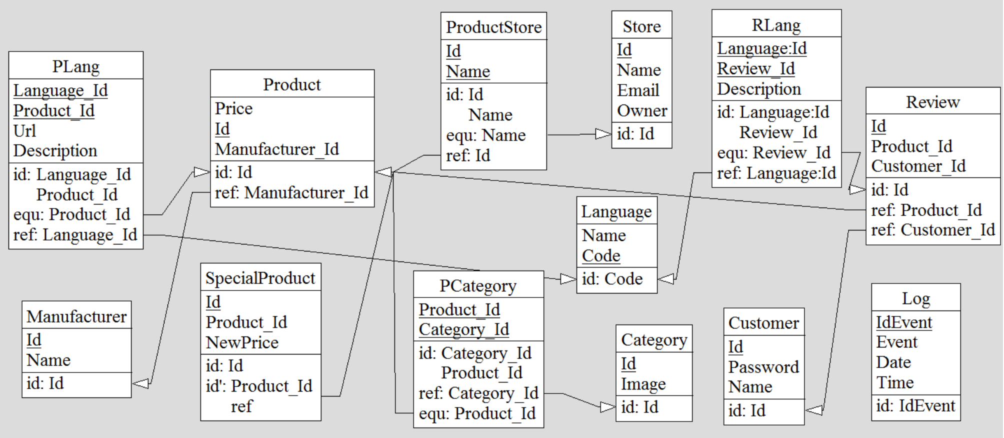


Figure 6.1: The logical schema (Web-Store case study)

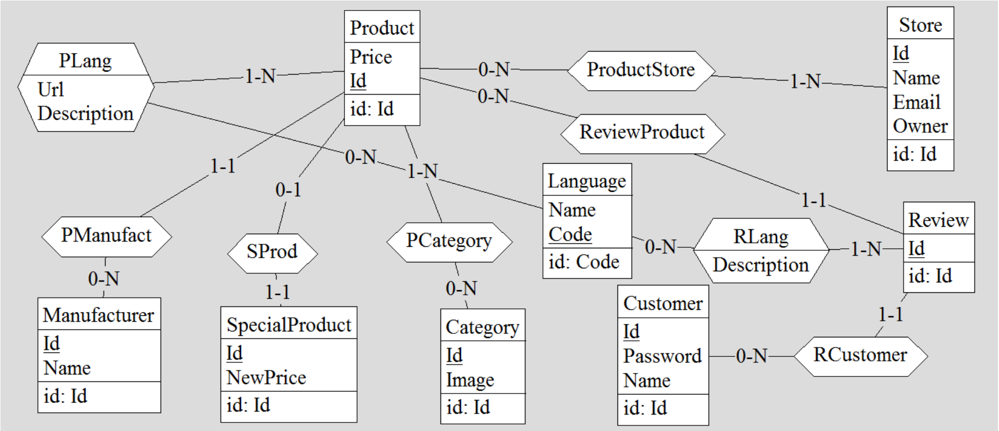


Figure 6.2: The conceptual schema (Web-Store case study)

Capturing and understanding the interactions of the Web Store with its database supporting the understanding of the feature processes, it supports conformance checking with respect to a contract model; and lastly it can also support the enhancement of the Web-Store processes. With the aim of illustrating how we extract processes from database interactions, we present an excerpt from a global Web-Store database in Figure 6.1 (logical schema) and Figure 6.2 (conceptual schema). In order to illustrate our approach, we will use an example of three SQL traces described in Figure 6.3.

Listing 6.1: Trace 1

```
q1: SELECT Customer.Password FROM Customer WHERE Customer.Id = ’Mark27’;
q2: SELECT Category.Id, Category.Image FROM Category;
q3: SELECT Product.Id, Product.Price FROM Product, PCategory  WHERE
    Product.Id=PCategory.Product_Id AND PCategory.Category_Id=’1’;
q4: SELECT PLang.Description FROM PLang, Language WHERE PLang.Language_Id=
    Language.Code AND PLang.Product_Id=’1A23’ AND Language.Name=’Italian’;
q5: SELECT SpecialProduct.NewPrice FROM SpecialProduct,Product WHERE
    SpecialProduct.Product_Id=Product.Id AND Product.Id=’1A23’;
q6: SELECT Manufacturer.Name FROM Manufacturer,Product WHERE Manufacturer.
    Id=Product.Manufacturer_Id AND Product.Id=’1A23’;
q7: SELECT PLang.Description FROM PLang, Language WHERE PLang.Language_Id=
    Language.Code AND PLang.Product_Id=’1F32’ AND Language.Name=’Italian’;
q8: SELECT SpecialProduct.NewPrice FROM SpecialProduct,Product WHERE
    SpecialProduct.Product_Id=Product.Id AND Product.Id=’1F32’;
q9: SELECT Manufacturer.Name FROM Manufacturer,Product WHERE Manufacturer.
    Id=Product.Manufacturer_Id AND Product.Id=’1F32’;
q10: INSERT INTO Log(IdEvent,Event,Date,Time) VALUES (’021’,’PrAcc1A23-1
    F32’,’2013-02-22’,’12:21:00’);
```

Listing 6.2: Trace 2

```
q11: SELECT Customer.Password FROM Customer WHERE Customer.Id = ’JennyMa’;
q12: SELECT Category.Id, Category.Image FROM Category;
q13: SELECT Product.Id, Product.Price FROM Product, PCategory WHERE
    Product.Id=PCategory.Product_Id AND PCategory.Category_Id=’2’;
```

Listing 6.3: Trace 3

```
q14: SELECT Customer.Password FROM Customer WHERE Customer.Id = ’DanWer’;
q15: SELECT Manufacturer.Id, Manufacturer.Name FROM Manufacturer ;
q16: SELECT Product.Id, Product.Price FROM Product WHERE Product.
    Manufacturer_Id=’AppleNamur01’ ;
q17: SELECT PLang.Description FROM PLang, Language WHERE PLang.Language_Id
    =Language.Code AND PLang.Product_Id=’2D11’ AND Language.Name=’Italian
    ’;
q18: SELECT SpecialProduct.NewPrice FROM SpecialProduct,Product WHERE
    SpecialProduct.Product_Id=Product.Id AND Product.Id=’2D11’;
q19: SELECT Manufacturer.Name FROM Manufacturer,Product WHERE Manufacturer
    .Id=Product.Manufacturer_Id AND Product.Id=’2D11’;
q20: INSERT INTO Log(IdEvent,Event,Date,Time) VALUES (’022’,’PrAcc2D11
    ’,’2013-02-28’,’14:00:03’);
```

Figure 6.3: Web Store: Three SQL execution traces

6.3 Approach and Research Questions

Here, we are going to define an approach that supports the extraction of the data-manipulation processes of data-intensive applications by analyzing *multiple* SQL execution traces. To this end, we used the SQL parser defined in *the trace capturing level* (see Section 5.3.2) of the previous chapter. Starting from the parsing results, Figure 6.4 describes the defined bottom-

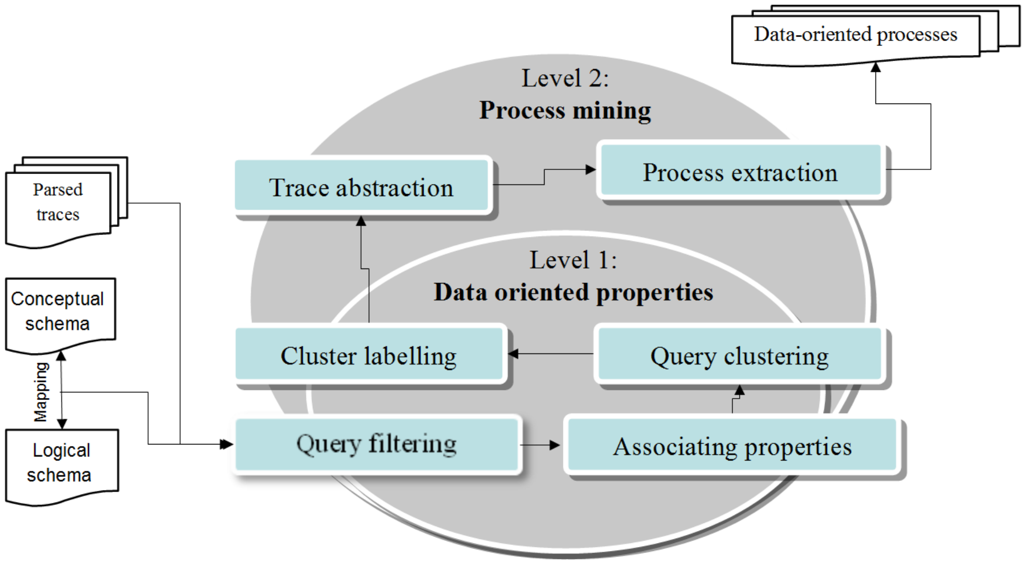


Figure 6.4: Bottom-up approach: Inter-scenario analysis.

up approach. The approach requires two additional inputs (a logical schema and a conceptual schema) and it examines two levels of understanding, these being: (1) *The data-oriented properties level* and (2) *the process mining level*, where each level contains a set of steps. These are:

- ***Level 1: (Data-oriented Properties)***

    1. **Query filtering** removes queries that do not refer to concepts and relationships of the input conceptual schema.
    2. **Associating properties** assigns to each query a set of data-oriented properties, each describing its data-manipulation behavior.
    3. **Query clustering** clusters queries that have the same set of properties, thus they implement the same data-manipulation function.
    4. **Cluster labeling** identifies the signature representing the data-manipulation function (i.e., cluster). This signature is expressed in terms of a label and a set of input/output parameters.

- ***Level 2: (Process mining)***

    1. **Traces abstraction**, seeks to replace traces of queries with their corresponding signatures.

2. **Process extraction**, generates the *data-oriented process* corresponding to the input traces of a feature.

The defined approach will allow us to successively address the following research questions for a given program execution scenario:

- **RQ1:** To what extent can we extract the data-oriented functions?
- **RQ2:** To what extent can we extract the data-manipulation processes?

6.4 Data-oriented Properties

Now, we will describe each step belonging to the first level of the approach. We are going also to illustrate each step by means of the illustrative scenario outlined earlier.

6.4.1 Query Filtering

Using the mapping between the conceptual schema and the logical schema (which consists of the correspondence relationship between each logical object and its corresponding conceptual object), we filter out from the input traces all the queries that do not express end-user concepts, i.e., the ones referring to database system tables or log tables that only appear in the logical schema.

Definition 6.4.1. *Suppose we have a set of parsed queries (PQ), the sub-logical schema (SL) and the sub-conceptual schema (SC). Then, we obtain a set of filtered queries (FQ), which satisfy the following condition:*

$$FQ = \{\ q\ | \exists objL \in SL,\ \exists objC \in SC\ : existmapping(objL, objC) \wedge objL \in q.objs \wedge q \in PQ\}$$

- $objL$ denotes a relational object (table or column) belonging to the sub-logical schema SL;
- $objC$ denotes a conceptual object (entity, attribute or relationship) belonging to the sub-conceptual schema SC;
- $q.objs$ denotes the set of objects used by q;

In our example (Figure 6.3), we remove q_{10} and q_{20} accessing table *Log* without a counterpart in the conceptual schema (see Figure 6.1 and Figure 6.2).

Query	Clauses	Properties

SELECT col1, col2 **FROM** tab1 **WHERE** col3 = val1;	**Selectclause** = { col1, col2 }
	Fromclause = { tab1 }
	Whereclause = {(=, col3 , val1) }
		p1 = SELECT tab1.col1 tab1.col2
		p2 = col3. EQ_value

SELECT tab1.col1, tab1.col2 **FROM** tab1, tab2 **WHERE** tab1.col1 = tab2.col1;	**Selectclause** = { tab1.col1, tab1.col2 }
	Fromclause = { tab1 , tab2 }
	Whereclause = {(=, tab1.col1 , tab2.col1) }
		p1 = SELECT tab1.col1 tab1.col2
		p2 = tab1.col1 = tab2.col1

INSERT INTO tab1 (col1, col2, ..., coln) **VALUES** (val1, val2, ..., valn) ;	**Insertclause** = { tab1 }
	Valuesclause = {(=, col1, val1), (=, col2, val2), ... (=, coln, valn)}
		p1 = INSERT INTO tab1

UPDATE tab1 **SET** col1 = val1, col2 = val2, ..., coln = valn **WHERE** col1= val1 ;	**Updatelause** = { tab1 }
	Setclause = {(=, col1, val1), (=, col2, val2), ... (=, coln, valn)}
	Whereclause = {(=, col1, val1)}
		p1 = Update tab1
		p2 = col1. EQ_value

DELETE FROM tab1 **WHERE** col1= val1 ;	**Deleteclause** = { tab1 }
	Whereclause = {(=, col1, val1)}
		p1 = DELETE tab1
		p2 = col1. EQ_value

Table 6.1: The rules of associating properties according to the parser clauses

6.4.2 Associating Properties

Using the queries parser (see Section 5.3.2) on the filtered traces, this step seeks to assign to each SQL statement a set of data-oriented properties according to the resulting clauses of the parsing phase. It is worth recalling that the parser just considers the recovered or modified information and the related selection criteria, and it ignores the actual values taken as input and produced as output by each query.

To this end, for each type of query, we defined the properties to record. For a *select* query, we record a property with the *select* clause. And for *delete*, *update* or *insert* queries, we record a property with the name of the table. If the query is *update*, we also record a property with the *set* clause and all its attributes. Lastly, for all query types except the *insert*, we add a property for the *where* clauses along with their attributes. Table 6.1 enumerates these rules through examples.

Table 6.2 shows the data-oriented properties of the three SQL traces presented in Figure 6.3. And Table 6.3 lists their corresponding properties. These queries are created starting from the logical schema represented in Figure 6.1. For instance, query q_1 is a select query over attribute *Password* of *Customer* table (property p_1) and it contains a *where* clause with an equality condition over *Id* attribute (p_2); query q_2 is a *select* query over attributes *Id* and the *Image* of *Category*, and it corresponds to property p_3; query q_3 is a *select* over attributes *Id* and *Price* of *Product* (property

Number	**Data-oriented property**
p1	SELECT Customer.Password
p2	Customer.Id.EQ_VALUE
p3	SELECT Category.Id Category.Image
p4	SELECT Product.Id Product.Price
p5	Product.Id=PCategory.Product_Id
p6	PCategory.Category_Id.EQ_VALUE
p7	SELECT PLang.Description
p8	PLang.Language_Id=Language.Code
p9	PLang.Product_Id.EQ_VALUE
p10	Language.Name.EQ_VALUE
p11	SELECT SpecialProduct.NewPrice
p12	SpecialProduct.Product_Id=Product.Id
p13	Product.Id.EQ_VALUE
p14	SELECT Manufacturer.Name
p15	Product.Manufacturer_Id=Manufacturer.Id
p16	INSERT INTO Log
p17	SELECT Manufacturer.Id Manufacturer.Name
p18	Product.Manufacturer_Id.EQ_VALUE

Table 6.2: Web Store: Properties of the traces of SQL statements

p_4), it contains two where clauses, i.e., a natural join between *Product.Id* and *PCategory.Product_Id* (p_5) and an equality condition over *PCategory .Category_Id* attribute (p_6).

6.4.3 Query Clustering

Starting with these properties, we used the FCA (see Def.5.5.5) in order to define a *property clustering algorithm* whose aim is to group together queries that implement the same data-manipulation function. It is worth noticing that FCA is much more powerful than how we used it earlier. Indeed, we did not consider the possibility of objects (queries) having only subsets of equal properties. Nevertheless, we implemented an algorithm to automate the clustering of queries having the same set of data-oriented properties. The algorithm is described in Listing 6.

In our case, each concept contains the queries that implement the same data-manipulation function, i.e. SQL statements of the same type and with equal input and output parameters, but with possible different values taken by the input and output parameters. We build a concept lattice starting

Query	Properties	Query	Properties
q1	p1,p2	q11	p1,p2
q2	p3	q12	p3
q3	p4,p5,p6	q13	p4,p5,p6
q4	p7,p8,p9,p10	q14	p1,p2
q5	p11,p12,p13	q15	p17
q6	p14,p15,p13	q16	p4,p18
q7	p7,p8,p9,p10	q17	p7,p8,p9,p10
q8	p11,p12,p13	q18	p11,p12,p13
q9	p14,p15,p13	q19	p14,p15,p13

Table 6.3: Web Store: Properties of the SQL statements

Algorithm 6 Properties clustering algorithm

```
Require: A set of parsed queries $PQ = \{q_1, ..., q_n\}$
Ensure: A set of clusters $C$
for all $q \in PQ$ do
    $PropQ \leftarrow AssociatProperties(q)$
    $Objects \leftarrow Add(q)$
    $Attributs \leftarrow Add(PropQ)$
end for
$C \leftarrow Clustering(Objects, Attributs)$
```

from a set of SQL statement having each a certain set of properties. Then, for each query, we search for the concept that has the same set of properties of the query. After, we divide the queries into disjoint sets (clusters), each performing different operations on the data.

We report in Table 6.4 the clusters obtained from queries in Table 6.3 by applying the defined algorithm. It shows the clustering results built from a sequence of SQL statements containing queries of Figure 6.3. Each cluster can be identified by the set of SQL statements implementing the same data-manipulation function and the properties that these queries have.

6.4.4 Cluster Labeling

Starting from these clusters, this step attempts to assign to each cluster a unique and meaningful label representing the data-manipulation function of the queries belonging to the cluster. For this, it requires an additional input, which is the conceptual database schema. We identify a signature in terms of a label and input/output (I/O), where the label is obtained by

Cluster	**Queries**	**Properties**
c_1	$\{q_1, q_{11}, q_{14}\}$	$\{p_1, p_2\}$
c_2	$\{q_2, q_{12}\}$	$\{p_3\}$
c_3	$\{q_3, q_{13}\}$	$\{p_4, p_5, p_6\}$
c_4	$\{q_4, q_7, q_{17}\}$	$\{p_7, p_8, p_9, p_{10}\}$
c_5	$\{q_5, q_8, q_{18}\}$	$\{p_{11}, p_{12}, p_{13}\}$
c_6	$\{q_6, q_9, q_{19}\}$	$\{p_{13}, p_{14}, p_{15}\}$
c_7	$\{q_{15}\}$	$\{p_{17}\}$
c_8	$\{q_{16}\}$	$\{p_4, p_{18}\}$

Table 6.4: Clusters of SQL queries of the Web Store application 6.3

exploiting information contained in the conceptual database schema, while I/O parameters are created based on the data-oriented properties belonging to each cluster.

The labels are obtained by analyzing the fragment of conceptual schema that corresponds to the logical subschema accessed by the clustered queries. In the case where the conceptual schema is not available, it is sufficient to reverse engineer the logical schema by adopting a data-modeling tool like DB-MAIN. Thus, given that the logical schema contains meaningful names, it is still possible to obtain significant labels.

To determine the labels, we extract the portion of conceptual schema accessed by the queries of a cluster and we apply a different labeling strategy according to the query types. As regards the query types *insert*, *delete* and *update*, we create the label of the data-manipulation functions starting from the unique entity E of the conceptual schema accessed by each of these query types, i.e., *InsertIntoE*, *DeleteFromE* and *UpdateE* respectively.

In the case of a *select* query, we distinguish four cases (Table 6.5) according to the portion of the conceptual schema involved in the clustered queries (see to Table 6.4 and Figure 6.2 for the given examples):

1. One entity E, in this case, we proposed two possible mapping names based on the presence of an equality condition over the primary key of E. If such a condition is present, we map the cluster with the label $getEById$. Conversely, we simply map the cluster with the label $getAllE$. In our example, we translate cluster *C1* to $getCustomerById$ since queries in *C1* retrieve information contained within entity $Customer$ by taking as input its primary key. Concerning cluster *C2*, since its queries retrieve all tuples of entity $Category$ without considering any condition over its primary key, it translates to $getAllCategory$.

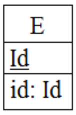
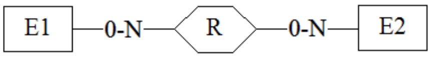
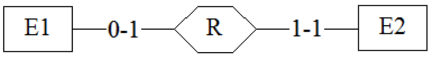
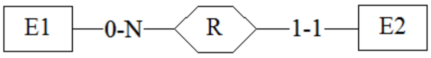
Case	**Label**
	- $getEById$ - $getAllE$
	- $getAllE_1OfE_2ViaR$
	- $getE_1OfE_2ViaR$
	- $getE_1OfE_2ViaR$ - $getAllE_2OfE_1ViaR$

Table 6.5: The rules of labeling for the select query

2. Two entities E_1, E_2 related by a many-to-many relationship R. In this case, the adopted label is $getAllE_1OfE_2ViaR$ providing that the queries give as a result the attributes of all the instances of E_1 associated with a given instance of E_2 through R. As for our example, we translate clusters *C3* to *getAllProductOfCategoryViaPCategory* since it extracts all the products of a certain category and we translate *C4* to *getAllLanguageOfProductViaPLang* provided that it extracts all the language descriptions of a product.

3. Two entities E_1, E_2 are related by a one-to-one relationship R. In this case, we map the cluster with the label $getE_1OfE_2ViaR$ provided that the queries retrieve the instance of E_1 associated to a certain input instance of E_2. In the Web-store example, we map cluster *C5* to label *getSpecialProductOfProductViaSProd* in order to extract the

occurrence of *SpecialProduct* related to a certain product via the one-to-one relationship $SProd$.

4. Two entities E_1, E_2 are related by a one-to-many relationship R. In this case, we distinguish two cases. If the queries return the instance of E_1 that participates to the relationship R with multiplicity N, we translate the query with the function $getE_1OfE_2ViaR$. Conversely, if the query returns the set of instances of E_2 that participate to R with multiplicity 1, we translate the query with the label $getAllE_2OfE_1ViaR$. In our Web-Store example, we translate cluster *C6* to *getManufacturerOfProductViaPManufact* since it retrieves the single occurrence of $Manufacturer$ participating to the relationship $PManufact$ with $Product$. We translate cluster *C8* to *getAllProductOfManufacturerViaPManufact* since it retrieves the multiple occurrences of $Product$ related to $Manufacturer$ via $PManufact$.

After the labeling phase, we also look for the input parameters taken by each data-manipulation function. To this end, we extract for each cluster the input parameters of the data manipulation functions, which consist of the attributes involved in equality or inequality conditions that appear in the data-oriented properties of the queries. For the output parameters, we simply list the set of attributes that are within of the property of the query starts with a select. Here, we also introduce a complex type in case several attributes are the outputs of a data-manipulation function. Table 6.6 shows the list of data-manipulation functions for the Web Store case study along with their input and output parameters.

Cluster	**Input**	**Output**
c_1:*getCustomerById*	$\{Id\}$	$\{Password\}$
c_2:*getAllCategory*	$-$	$\{Id, Image\}$
c_3:*getAllProductOfCategoryViaPCategory*	$\{Category\_Id\}$	$\{Id, Price\}$
c_4:*getAllLanguageOfProductViaPLang*	$\{Product\_Id, Name\}$	$\{Description\}$
c_5:*getSpecialProductOfProductViaSProd*	$\{Product.Id\}$	$\{NewPrice\}$
c_6:*getManufacturerOfProductViaPManufact*	$\{Product.Id\}$	$\{Name\}$
c_7:*getAllManufacturer*	$-$	$\{Id, Name\}$
c_8:*getAllProductOfManufacturerViaPManufact*	$\{Manufacturer\_Id\}$	$\{Id, Price\}$

Table 6.6: Web Store: Data manipulation functions and I/O parameters

It is worth noticing that in our approach it is enough to translate just one arbitrary query within the same cluster and to evaluate its input and output parameters; indeed all queries belonging to a cluster have the same set of properties and they consequently access the same portion of the conceptual schema. In defining signatures, we have not considered the complete SQL grammar, e.g., we ignored *group by* operators that add more fined-grained information at the attribute level and we ignored the *where* clauses without value-based equality and inequality conditions. Nevertheless, we plan to adopt the latter so as to provide more detailed definitions of the data manipulation functions.

6.5 Process Mining

For this level, we explain how it is possible to generate the data-oriented process corresponding to the input traces. After replacing queries with their corresponding meaningful signatures, this level seeks to extract a process model. Such a process model may be the basis for supporting a different analysis, namely checking the conformance of the process model defined at run-time, enhancing it in case it is not compliant with the one defined at design-time (if it exists) and more in general, and it forms the basis for re-documenting the data-manipulation behavior of an already implemented system.

So as to be able to generate a process starting from the traces of data-manipulation functions of the previous level (data-oriented properties), we will consider two steps. These are: (1) *The traces abstraction*, which replaces SQL traces with their corresponding data-manipulation functions. (2)*Process extraction*, which exploits a process-mining algorithm to extract the feature behavior as a sequence of function executions with sequential, parallel and choice operators.

6.5.1 Traces Abstraction

The objective of this step is to abstract the trace given as an input by replacing each SQL statement with the corresponding meaningful signatures. As a result, we get a new trace containing successive data-manipulation functions corresponding to the given program execution scenario. Listing 6.5 depicts the abstracted traces of the three traces given in the listening 6.3, where:

- ***Trace 1:*** It gets customer information (*C1*), and it performs a category-driven search of products by means of getting all the product categories (*C2*) and all the products of a certain selected category (*C3*). For each retrieved product, three functions are twice iterated: *C4* retrieves product description, *C5* extracts special product information and *C6* extracts related manufacturer information.

Listing 6.4: Abstracted Trace 1

```
q1: getCustomerById(C1)
q2: getAllCategory(C2)
q3: getAllProductOfCategoryViaPCategory(C3)
q4: getAllLanguageOfProductViaPLang(C4)
q5: getSpecialProductOfProductViaSProd(C5)
q6: getManufacturerOfProductViaPManufact(C6)
q7: getAllLanguageOfProductViaPLang(C4)
q8: getSpecialProductOfProductViaSProd(C5)
q9: getManufacturerOfProductViaPManufact(C6)
```

Listing 6.5: Abstracted Trace 2

```
q11: getCustomerById(C1)
q12: getAllCategory(C2)
q13: getAllProductOfCategoryViaPCategory(C3)
```

Listing 6.6: Abstracted Trace 3

```
q14: getCustomerById(C1)
q15: getAllManufacturer(C7)
q16: getAllProductOfManufacturerViaPManufact(C8)
q17: getAllLanguageOfProductViaPLang(C4)
q18: getSpecialProductOfProductViaSProd(C5)
q19: getManufacturerOfProductViaPManufact(C6)
```

Figure 6.5: Web Store : Abstracted Traces of SQL statements

- ***Trace 2:*** It is different from *Trace 1*, because after function C3 no products are retrieved and the process ends.

- ***Trace 3:*** This is similar to *Trace 1* except that it searches products based on their manufacturer (functions *C7* and *C8*) instead of searching by category (*C2* and *C3*).

6.5.2 Process Extraction

We extract the data-manipulation behavior in terms of a Petri Net [65], which consists of a set of *places*, *transitions*, *directed arcs* and *tokens*. In order to describe this step, we need to formally define this concept in order to be able to understand the process.

Definition 6.5.1. *Petri nets are a basic model of parallel and distributed systems. The basic idea is to describe state changes in a system with transitions. According to the definition given by Murata et al. [55], Petri net consists of two components, namely a net structure and an initial marking. A net structure contains two sorts of nodes, namely places and transitions.*

Trace 1	Trace 2	Trace 3
START getStoreById (C1) getAllCategory (C2) getAllProductOfCategoryViaPCategory (C3) getAllLanguageOfProductViaPLang (C4) getSpecialProductOfProductViaSProd (C5) getAllLanguageOfProductViaPLang (C6) getSpecialProductOfProductViaSProd (C4) getManufacturerOfProductViaPManufact (C5) getLanguageOfReviewViaRLang (C6) END	START getStoreById (C1) getAllCategory (C2) getAllProductOfCategoryViaPCategory (C3) END	START getCustomerById (C1) getAllManufacturer (C7) getAllProductOfManufacturerViaPManufact (C8) getAllLanguageOfProductViaPLang (C4) getSpecialProductOfProductViaSProd (C5) getManufacturerOfProductViaPManufact (C6) END

Table 6.7: Example traces

Arcs run from a place to a transition or vice versa, never between places or between transitions. More formally, according to the definition given by Zuberek [95], a Petri Net is a 5-tuple N=$\{P, T, A, w, B\}$ *where:*

- P *is a non-empty finite set of places*
- T *is a non-empty finite set of transitions*
- A *is a set ot directed arcs which connects places with transitions and transitions with places,* $A \subseteq P \times T \cup T \times P$
- w *is a weight function which assigns a positive integer "weight" to each arc of the net,* $w : A \rightarrow \{1, 2, ...\}$
- B *is a (possibly empty) set of inhibitor arcs,* $B \subseteq P \times T$*, and* A *and* B *are disjoint sets*

By adapting this definition to our traces Web-Store motivating scenario, we create a process model for the feature represented by the traces, which implements the visualization of products along with basic products details, manufacturer information and if present authors' reviews. In order to produce the process model, we consider as input a set of process instances, each expressed as a sequence of data-manipulation functions generated from the previous step. Starting from sequences of SQL statements, each of these sequences corresponds to a different instance of the process performed by a single user during a certain session and they can be captured by monitoring database access. Table 6.5.2 shows the three possible instances of the process, each with an additional starting function and ending function.

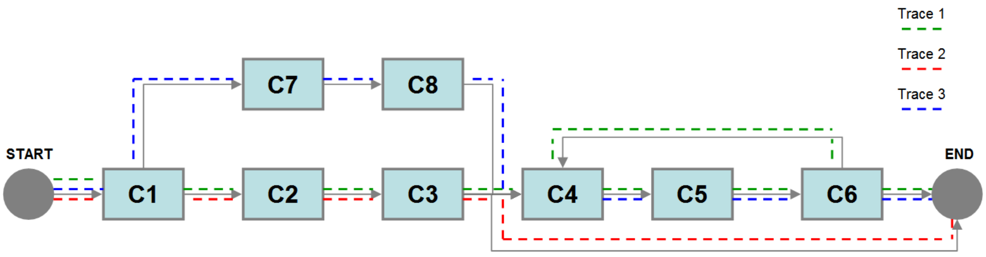


Figure 6.6: Web Store: Process mined with *Trace 1*, *2* and *3*.

Figure 6.6 shows the process model mined starting from the three traces in Table 6.5.2. The process contains a beginning ($START$) and an ending (END) task since they are always present within the input traces. After the $START$ task, the process performs the first function $getStoreById(C1)$.

Then, the process has two possibilities, these being: (1) The two functions $getAllManufacturer(C7)$ and $getAllProductOfManufacturerViaPManufact(C8)$ are performed. Then, the process performs the three functions $getAllLanguageOfProductViaPLang(C4)$, $getSpecialProductOfProductViaSProd(C5)$ and $getManufacturerOfProductViaPManufact(C6)$, before the task END is performed and the process is terminated (Trace 3); (2) The two functions $getAllCategory(C2)$ and $getAllProductOfCategoryViaPCategory(c3)$ are performed. Then, the process has a decision point; either task END is performed and the process is terminated (Trace 2) or it enters in a loop, where functions $getAllLanguageOfProductViaPLang(C4)$, $getSpecialProductOfProductViaSProd(C5)$ and $getManufacturerOfProductViaPManufact(C6)$ are iterated, before the task END (Trace 1).

It is worth noticing that if we look more carefully at the traces and the process together, we see that even if the process is able to re-produce the traces, it can also generate additional traces. This depends on the chosen process miner algorithm that produces complete models at the price of introducing noise. Our goal is to be able to recover the model that reproduces all the input traces, while we do not take into account appropriateness, meaning that the model can generate a bigger set of traces than the input one. These requirements will guide our selection of the best process miner to adopt during the experimentation phase.

6.6 Evaluation

The approach presented in this chapter is implemented in an integrated tool, which takes as input a set of SQL traces (each representing an instance of the same feature), the logical schema and optionally the conceptual schema and the mapping between both. The tool relies on a set of implemented components.

Among the implemented components, we exploited some components that were implemented by DAViS 5.7, namely: (1) the *SQL parser* component (Section 5.3.2), we exploited it in order to extract the data-oriented properties; (2) the *filtering* component (Section 5.4.1), and we used the conceptual subschema in order to filter out the SQL queries referring to objects that are not in the input conceptual schema; (3) the *clustering* component 5.5.2 to cluster queries according to their properties.

Based on the previous components, the *labeling* component generates data-manipulation functions (i.e., cluster signatures). And the *traces abstraction* component uses a Java library [1] to create standardized event logs.

[1] http://www.xes-standard.org/openxes/start

Starting from the event logs obtained from the previous component, we rely on the standard open source Process Mining framework (ProM) [86] to mine data-manipulation processes. *ProM* integrates the functionality of several existing process-mining tools and provides many additional process-mining plug-ins. It supports multiple formats and multiple languages, such as Petri nets and Social Networks. In the context of this approach, we chose as a miner algorithm the Integer Linear Programming (ILP) miner [85], since it is able to produce complete process models *(i.e., Petri Nets [65])* with a low level of noise. Petri nets, which come from the ILP miner, are semantically well-defined models that permit different types of analysis, among which is a precise comparison of different model instances. Once a process has been created, we exploit the ProM tool to export the Petri Net as a Petri Net Markup Language (PNML) – as a proposal of an XML-based interchange format for a Petri net file, which can be given as input to a Petri Net editor tool, e.g., *WoPeD*[2] allowing reading and editing operations. ProM provides user-friendly graphical interfaces which support designers in easily loading standardized event logs, mining Petri Nets through ILP miner and exporting such models for editing purposes enabled by WoPeD.

6.6.1 Scenario Used

In order to validate the proposed approach, we collected a set of database access traces from an e-restaurant Web application developed by one of our undergraduate students at our university.

This data-intensive system provides different features, each accessing a different portion of an underlying database to support the activities of taxi drivers, restaurant owners and clients. Clients consult an information menu (*MenuInfo* feature), information about special offers (*DailySpecials* feature) and general information about restaurants (*RestaurantInfo* feature). They can reserve a table for a meal (*Reservation* feature) and they can issue orders of meals with two possible options: they either pick up the meal at the restaurant or they ask for a taxi service to deliver the booked meal to them (*IssueOrders* feature). Restaurant owners prepare the meal to be delivered to clients (*RestaurantOrders* feature), while taxi drivers check delivery requests and they carry out the delivering process (*TaxiDelivery* feature).

Among a wide set of implemented features, we chose a subset of features whose data-manipulation processes covered execution patterns of a different nature, e.g., sequential execution, cycles and decision points. Since we played the role of the designer, we were able to select the most interesting features

[2] www.woped.org

i.e., *DailySpecials*, *RestaurantInfo*, *MenuInfo*, *Reservation* and *IssueOrders*. Consequently, we collected the corresponding sequences of SQL queries to give as input to the tool. We assume that for each feature, its input set of extracted traces corresponds to a 100% coverage ratio of the process and it contains exactly 6 different traces.

6.6.2 Experiments Description

Starting from the input data, we conducted a set of experiments seeking to answer the following research question: *How good are the processes extracted through the integrated tool with a variable trace coverage, with respect to their correct versions identified by the designer?*

To address this research question, we decided to divide the experiment in two different phases:

- The *start-up* phase creates for each feature the correct processes starting from its complete set of traces (trace coverage = 100%) with the support of the integrated tool and the intervention of the designer;

- The *core* phase mines processes with a variable trace coverage ($\leq$ 100%) and it evaluates the goodness of such processes with respect to the correct ones previously identified with the support of ProM tool plug-ins.

6.6.2.1 Start-up

For each feature, we adapted the tool so as to create the data-manipulation processes with the complete set of traces. Consequently, since these models may either contain noise or they may not be complete, we asked the designer to derive a correct version from it. It is worth pointing out that the designers have in dept knowledge and understanding of the processes and they can easily assess the system regardless of whether a certain process is correct or not.

In our case, as designers, we adapted the WoPeD tool in order to visualize the processes as Petri Nets and to perform the possible required modifications, i.e., addition/deletion of places and arcs. For instance, Figure 6.7 (top) shows the feature *RestaurantReservation* mined with the proposed approach, while Figure 6.7 (bottom) shows the version of the same feature as it had been corrected by the designer.

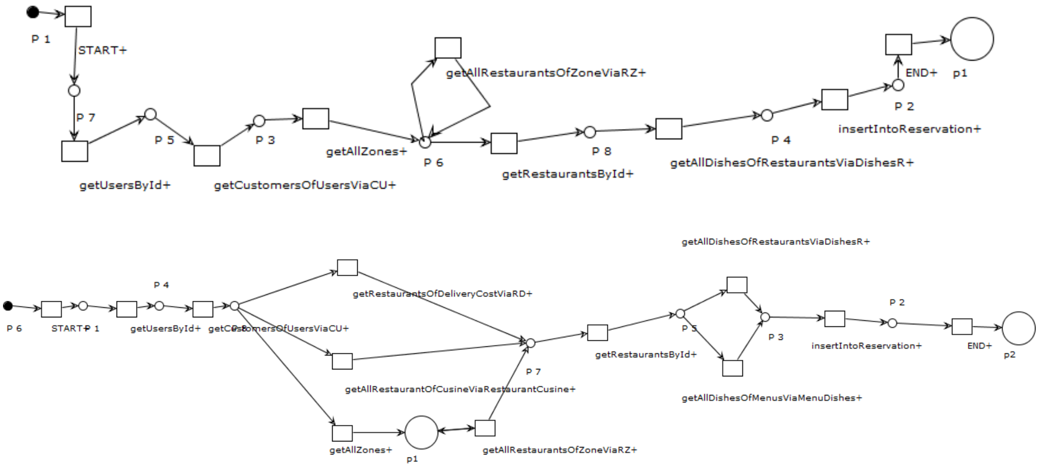

P 1
START+
P 7
getUsersById+
P 5
getCustomersOfUsersViaCU+
P 3
getAllZones+
getAllRestaurantsOfZoneViaRZ+
P 6
getRestaurantsById+
P 8
getAllDishesOfRestaurantsViaDishesR+
P 4
insertIntoReservation+
P 2
END+
p1
P 6
START+
P 1
getUsersById+
P 4
getCustomersOfUsersViaCU+
P 8
getRestaurantsOfDeliveryCostViaRD+
getAllRestaurantOfCusineViaRestaurantCusine+
getAllZones+
p1
getAllRestaurantsOfZoneViaRZ+
P 7
getRestaurantsById+
P 5
getAllDishesOfRestaurantsViaDishesR+
getAllDishesOfMenusViaMenuDishes+
P 3
insertIntoReservation+
P 2
END+
p2

Figure 6.7: *RestaurantReservation*: Mined process (top) and process corrected by the designer (bottom).

6.6.2.2 Core

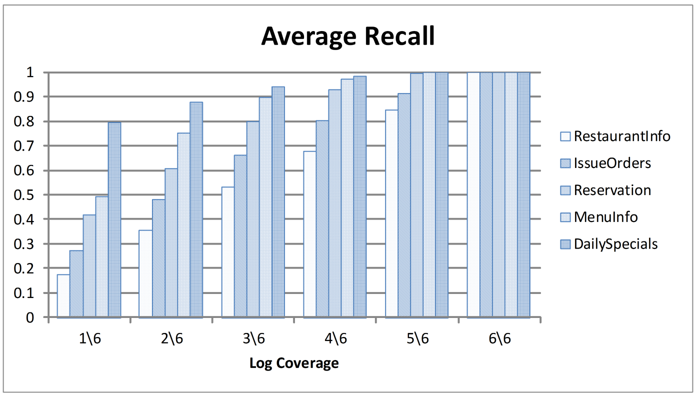
Once the designer has determined the correct versions of the processes of the input features, we performed a set of experiments in order to compare them with the models produced by our approach with a different ratio of the coverage of the input traces. To this end, we defined: (1) P_{tool} as the Petri Nets produced by our approach, (2) P_{exp} as the correct Petri Nets and (3) their corresponding set of valid traces as $T(P_{tool})$ and $T(P_{exp})$. After, we defined two metrics:

1. **Recall** $= \frac{|tp|}{|tp+fn|} = \frac{|T(P_{tool}) \cap T(P_{exp})|}{|T(P_{exp})|}$
2. **Precision** $= \frac{|tp|}{|tp+fp|} = \frac{|T(P_{tool}) \cap T(P_{exp})|}{|T(P_{tool})|}$

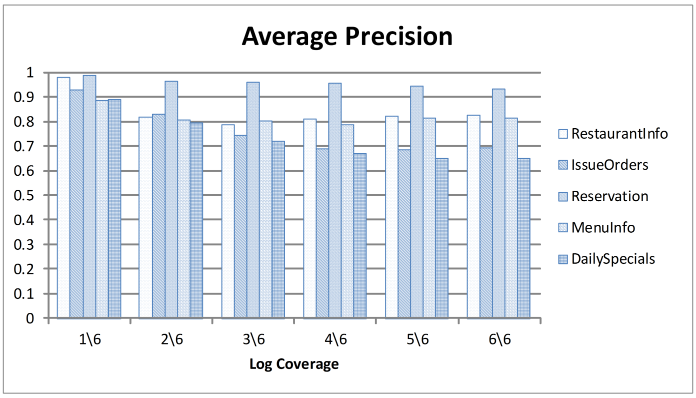
Here *tp* represents the true positive, which is the set of traces identified by our algorithm that is also included in the correct model, *fp* represents the false positive, which is the set of traces identified by our algorithm that have not been included in the correct model, while *fn* represents the false negative, which is the set of traces not identified by our algorithm but it has been included in the correct model.

Given that the set of traces for a Petri Net could be infinite, we consider as an approximation the minimum number of traces that can cover the Petri Net, where loops are iterated at most once. Since the ProM tool already provides a plug-in for evaluating precision and recall between two Heuristic

Net's and to translate a Petri Net to a Heuristic Nets, we adopted both of them for evaluating mined models.



(a)



(b)

Figure 6.8: e-Restaurant case study: Average recall measure (a) and average precision measure (b) of the mined process models depending on log coverage (1-trace logs, 2-trace logs, ..., 6-trace logs).

For each feature, we mined different processes starting from a variable set of input traces. We considered as input all the combinations of $1, ..., t$ traces from the global set of t traces (in our experiments t=6). For each combination of traces, we mined the corresponding process model and we measured precision and recall values of this model relative to the correct one. Finally, we evaluated the average precision and recall obtained with all the combinations of 1 trace from the t traces, all the combinations of 2 traces from t traces, ..., and all the combinations of t traces from t traces. Then, we repeated the same set of experiments for all the features. Figures 6.8(a) and

6.8(b) show the average recall and precision values obtained for the input features.

Figure 6.8(a) tells us that for all processes the average recall measure increases if we consider a greater set of traces as input. When we consider all the traces, we have a recall equal to 1, meaning that all the traces within the correct model have been identified by our approach. On average, if the number of traces decreases, the recall decreases as well.

In Figure 6.8(b), the average precision measure increases with the reduction of traces considered as input, meaning that on average with a lower number of traces the noise introduced by our approach is lower than that with a greater number of traces. We claim that the trend of precision and recall averages does not depend on the nature of processes; indeed, they all follow the same behavior that depends on the coverage of the input log.

6.6.3 Threats To Validity

Our approach extracts data-manipulation process models that may suffer of two different types of noise:

- The first belongs to the adapted mining algorithm and it results in incorrectly mined models having possibly additional traces. To mitigate this noise problem, we asked designers to perform a correction to the models mined by our approach. In this way, we were able to create models that were 100% correct and which had been exploited as input to evaluate the sensitivity of our approach depending on the log coverage.

- The second type of noise belongs to the increasing number of SQL statements that refer to technical implementation details not relevant for the application logic. To overcome this problem, we introduced a preprocessing phase to filter out queries that do not refer to a certain subset of the conceptual schema as selected by the designer.

Our approach may also have threats to scalability depending on the increasing input of SQL statements. Indeed, even after the non-relevant queries have been pruned, we could still mine a non-readable process due to the large space of extracted data-manipulation functions. To tackle this problem, we advised designers to prune iteratively the conceptual schema until they obtained a readable process (by iteratively applying our approach to the input set of queries).

Recall and precision values obtained for the different features depend on the adapted mining algorithm. In [4], different mining algorithms were compared to identify the one that better fits the application needs. In our experiments, we exploited the ILP miner algorithm, which is able to create models with a fitness (precision) of 100% and an acceptable level of appropriateness (recall) with respect to the input set of traces. By adapting mining algorithms with different fitness and appropriateness, we would have obtained different precision and recall values. Nevertheless, we claim that we expect to get similar trends of precision and recall depending on the log coverage. Indeed, we expect that by increasing the log coverage, we will subsequently increase the recall while lowering the precision.

The e-restaurant system adapted in the experiments can be considered a good representative for data-intensive systems i.e., systems where most of the complexity is concealed in its interactions with its database. Actually, the e-restaurant consists of numerous interactions with its database, where data form the basis for supporting the core business activities. Even though the experimental system is of limited size and complexity, it demonstrated sufficiently the capability of the proposed approach by revealing heterogeneous data-manipulation processes in real environments. With this in mind, we also mitigated the quality of the input traces by analyzing features of a different nature. However, we may consider the threat that the approach has been evaluated on a single case study (the e-restaurant system).

6.7 Conclusions

In this chapter, we proposed an approach for supporting the recovery of the data-manipulation processes of data-intensive systems via the analysis of multiple SQL execution Traces. The approach includes two phases. These are: (1) *Data-oriented properties:*, which seeks to extract data-manipulation functions and (2) *Process mining:*, which seeks to extract data-manipulation processes. Then, we applied the *ProM* and the *ILP* miner algorithms to extract the data-oriented processes of an e-restaurant Web application and we conducted a set of preliminary experiments to assess the sensitivity of our technique in producing correct processes that depend on the traces of the log coverage.

Chapter 7

Code Re-documentation from SQL Execution Traces

The next best thing to knowing something is knowing where to find it.

Samuel Johnson

Contents

7.1 Introduction 129

7.2 Motivation 130

7.3 Approach and Research Questions 131

7.4 Preliminaries 132

7.5 Comments Generation 135

7.6 Comments Injection 138

7.7 Discussions 139

7.8 Illustrative Example 141

7.9 Results 143

7.10 Conclusions 144

7.1 Introduction

During software maintenance and evolution, software artifacts are constantly changing (e.g. source code, database schemas and software documentation). In this context, numerous studies have been interested in the co-evolution of the program's source code and its database schemas [9, 23, 43] at the expense of software documentation. As a consequence, an inherent gap has been created between software system and its documentation, due to the lack of an up-to-date documentation, which makes the software understanding task more difficult. This is why, in the context of software understanding, software documentation is a key ingredient, where improving software documentation helps to facilitate the software understanding task and, in turn, it helps software maintenance and evolution.

Assuming there is a lack of documentation, the approach we are going to define in this chapter seeks to reduce this gap by re-documenting the program's source code with the interpretation of data-manipulation behavior previously extracted through the analysis of SQL execution traces. Knowing that source code comments may be viewed as an another source of documentation could help developers to better understand nuances of database usage and the programs in some ways as well.

The chapter is structured as follows. We start by presenting the context and the motivation that led us to propose this approach. Then, we present some formal definitions of techniques that are related to our methodology before elaborating on the approach. Finally, to illustrate the approach we present an example and we demonstrate its feasibility.

7.2 Motivation

As explained earlier, data-intensive systems manipulate dynamically and intensively a huge amount of data stored in a database, which means the database will have a central place. Therefore, the understanding of such systems not only necessitates a detailed and an up-to-date knowledge of database schemas, but also documentation on how features are implemented by relying on database operations. Unfortunately, such information is not available for most if not all of the current systems. Linares-Vàsquez et al [45] estimated that 77% of the 33K+ studied methods of 3.1K+ open source Java projects invoking SQL statements were completely undocumented.

In the light of this, up-to-date source code comments about data-manipulation behavior may be viewed as a valuable source of documentation, especially for these kinds of systems. In this context, Linares-Vasquez et al. [44, 41] defined DBScribe, an approach that seeks to statically analyze source code and the database schema of a given database-centric application in order to automatically generate up-to-date natural language descriptions of the SQL statements related to methods' execution, along with the schema constraints imposed on these statements.

However, we can clearly highlight some limitations; e.g. the authors only treated the logical level as a basis for comments. For instance, «It inserts the

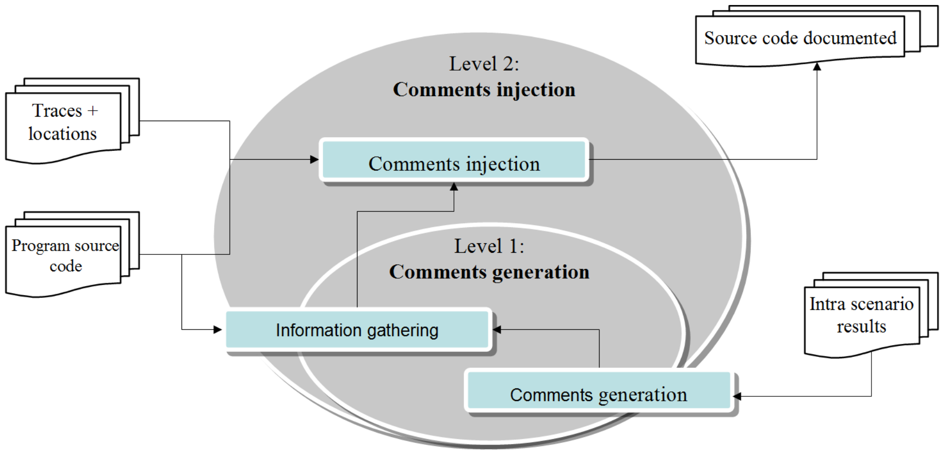


Figure 7.1: Bottom-up approach: A code re-documentation

Username, *Passwd* attributes into table *logindetails*» is a comment example of an insert query. Besides this, as we mentioned in Chapter 2, the logical level is usually highly technical, where sometimes objects have a technical name (e.g. cl_id). Here in this chapter we are going to define an approach for re-documenting program source code by treating the conceptual level as a basis for comments, i.e. comments will be expressed in terms of a domain-specific, platform-independent model.

In addition, given the dynamic nature of the interactions (programs/-database) of such systems, SQL statement are generally dynamically constructed preventing them to be easily analyzed by static analysis techniques. Hence, we will propose a new approach to generate comments according to the results obtained from the dynamic analysis of data-manipulation behavior and inject them into their corresponding source code location.

7.3 Approach and Research Questions

In the light of the above motivation, we defined a bottom-up approach which is depicted in Figure 7.1. This approach first attempts to automatically generate an up-to-date natural language interpretation of database-manipulation behavior. Then, it injects the generated interpretation as comments into the appropriate source code locations. This leads us to formulate the following research question: *To what extent can we re-documented the program source code starting from its SQL execution traces?*.

Starting from the dynamic analysis results of SQL execution traces obtained from the approach defined in Chapter 5, the approach includes two levels. These are:

1. ***Level 1: (Comments generation)*** this level seeks to answer the question about *what relevant information should be injected.* To this end, we first identify what type of information should be saved. Second, we generate comments to be injected according to the information type.

2. ***Level 2: (Comments injection)*** this level seeks to answer the question about *where should we inject the generated comments in program source code.* For this, we first identify the source code location before injecting the generated comments.

7.4 Preliminaries

Before introducing in detail each level of the approach, it is worth defining some techniques and concepts that we used in order to be able to understand what follows.

7.4.1 Java DataBase Connectivity (JDBC)

JDBC is an API Java (a set of classes and interfaces) defined in order to allow access to the relational databases using the Java language via SQL queries. JDBC allows Java applications to manage three main programming activities. These are:

- Connection to the database.

- Send queries and update statements to the database.

- Retrieve and process the results received from the database in response to the sent query.

Listing 7.4.1 depicts a Java code fragment using JDBC API to connect to the database, execute a SQL query and get the result of the query. The first step consists of instantiating a *DriverManager* object in order to establish a connection to the database driver and then to log into the database. The second step consists of instantiating a *Statement* object that carries the SQL language query to the database. Finally, the third step instantiates a *ResultSet* object in order to collect the results of the SQL statement.

Listing 7.1: Java code fragment using JDBC technology

```
public void DatabaseAccesses(String username, String password) {

// Connexion to the database
Connection connexion =
   DriverManager.getConnection("jdbc:myDriver:myDatabase",
   username, password);

// Querying the database
Statement statement = connexion.createStatement();
ResultSet result = statement.executeQuery("select name, ncust
   from customer where locality = ’Namur’");

// Retrieve the results received from the database
while (result.next()) {
String name = result.getInt("name");
String ncust = result.getString("ncust");
}
}
```

7.4.2 Extraction Techniques of SQL Trace Locations

In the literature, there is a surprisingly big number of techniques available for extracting and locating SQL execution statements that occur in a application program. They range from simply using a JDBC logger for instrumenting program source code to sophisticated program analysis techniques. Among these techniques, we shall briefly mention two of them:

7.4.2.1 JDBC Logger

The term logging consists of adding processing in applications in order to allow the storage of messages following events. Logging is used for different purposes and it provides, for instance, the means to trace or debug application programs and generate detailed log output, while keeping track of the exceptions that occur in the application program and the various events related to the execution of applications, etc. The JDBC driver provides a logging feature, which is a set of classes that allow logging information about events that occur when the JDBC driver code runs. JDBC logging feature can include either user-visible events, such as SQL exceptions, running of SQL statements or JDBC internal events, such as entry to and exit from

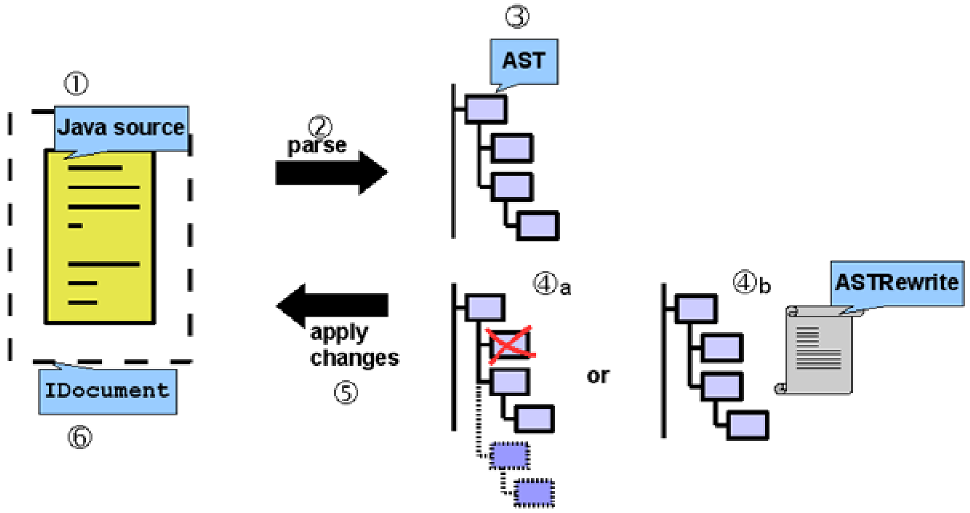


Figure 7.2: The Abstract Syntax Tree workflow [38]

internal JDBC methods.

7.4.2.2 Static Program Instrumentation

Using static program instrumentation 2.8.1, Loup et al. [51] defined a static analysis technique that sought to identify the dynamic database access locations and the database objects (tables and columns) accessed by a given access. They were interested in three types of Java database access technologies, these being JDBC, Hibernate and Java Persistence API (JPA). The authors identified the source code locations querying the database based on the call graph of the given method, which is derived from an inter-procedural analysis.

7.4.3 JAVA Abstract Syntax Tree (AST)

The Abstract Syntax Tree seeks to represent the abstract syntactic structure of source code based on a tree representation. Here, each node of the tree represents a construct occurring in the source code. The syntax is called «abstract» because it does not represent every detail appearing in the real syntax. According to the definition given by Kuhn [38], the AST is the base framework for many powerful tools of the Eclipse IDE such as refactoring, Quick Fix, Quick Assist. Therefore, the abstract syntax tree-based representation is very popular in studies that attempt to analyze and modify the source code.

Figure 7.2 describes a typical workflow of an application using an AST: (1) in the first step we provide the source code to parse; (2) then, the source code is placed in the parser; (3) the Abstract Syntax Tree is the output provided by the parser; (4) the fourth step consists of manipulating the AST (modifying, removing or adding) to achieve the chosen goal; (5) finally, changes are applied to the source code provided.

7.5 Comments Generation

7.5.1 Information Determination

As stated in the approach definition, our main goal is to re-document the program source code by adding comments about its data-manipulation behavior. To achieve this, it is important to identify what type of information should be considered and how it should be presented. For this purpose, we looked at two categories of information, namely (1) comments about implicit dependencies occurring between successive SQL queries; and (2) comments about the conceptual interpretation of SQL execution traces expressed in natural language terms.

7.5.1.1 Comments about Dependencies

As we said earlier, data-intensive systems are characterized by intensive interactions between programs and their database. Hence, understanding the behavior of their data-manipulation should provide a major support for the understanding of such systems. In the same way, we examine the issue of documenting program source code with comments about how SQL queries depend on each other and it offers a significant added-value to developers, especially in the case of systems where most of their methods invoke SQL statements.

For instance, let us assume that we have two method fragments ($MF1$, $MF2$), where each of them accesses the database via a SQL query $q1$ (respectively $q2$). In addition, there is a dependency between $q1$ and $q2$. In this case, adding a comment about this dependency before each query execution can be viewed as useful. This is especially helpful before making any changes (modification or deletion) on the two methods, or objects belonging to both queries. Otherwise, failing to correctly adapt programs or database may create program inconsistencies, which in turn may cause program failures.

For this purpose, based on the analysis results of the inter-queries level (see Section 5.5), we looked at two types of dependency, namely (1) output/input dependency; and (2) loops (nested queries).

Type	**Comment**
O/I	Output/Input [lx - ly]: tabx.colx ↔ taby.coly
Loops	Loop [lmq - lnq]: tabm.colm ↔ tabm.colm

Table 7.1: Dependency comment template

Table 7.1 gives the comment template of each of these two types. Here, each comment consists of three parts, these being (1) the type of the dependency (output/input, loop); (2) a pair [lx - ly], where *lx* and *ly* designate the source code location of the queries in question; and (3) the last part contains the dependent database objects.

Listing 7.2: A fragment of an example of a SQL trace

```
q1: select name, ncust from customer where Locality = 'Namur';
q2: select norder, date from order where ncust= 'B062';
q3: select norder, date from order where ncust= 'C123';
q4: select norder,  date from order where ncust= 'L422';
q5: select norder,  date from order where ncust= 'S127';
...
```

For instance, Listing 7.2 shows a fragment of five SQL queries representing a loop (nested queries), where the main query is $q1$ and $q2$ to $q5$ are the nested queries extracted from the system described in Section 5.3.1. And Listing 7.5.1.1 contains a portion of the JAVA source code corresponding to the trace of Listing 7.2.

Once we have analyzed the trace, the detected loop will be commented as: *Loop [8-14]: costumer.ncust ↔ order.ncust.*

7.5.1.2 SQL Interpretation Comments

The second category of comments concerns the natural language interpretation generated from the analysis of SQL execution traces. It is worth recalling that we got the natural language interpretations of SQL statements from *«the query interpretation level»* (see Section 5.6). Here, the latter provides as an output a set of natural language sentences expressing the conceptual interpretation of the trace given as input.

Injecting these sentences into the appropriate source code locations may be viewed as a support for developers and help them to better understand data-manipulation behavior, and also the relevant portion of program source code.

Listing 7.3: The program source code invoking the trace of Listing 7.2

```
...
public void SearchOrders(String nametab, String locality)
   throws SQLException{
...
this.connect();
String sqlselect="select ncust, name from "+nametab+" where
   Locality = ?";
java.sql.PreparedStatement preparedStatement1 =
   con.prepareStatement(sqlselect);
preparedStatement1.setString(1, locality);
ResultSet res = preparedStatement1.executeQuery(sqlselect);
while (res.next()) {
String ncust = res.getString(1);
String sqlselect2 = "select norder, date from order where
   ncust=?";
java.sql.PreparedStatement preparedStatement2 =
   con.prepareStatement(sqlselect);
preparedStatement2.setString(1, ncust);
ResultSet res2 = preparedStatement2.executeQuery(sqlselect2);
...
}
...
}
...
```

Listing 7.4: The natural language interpretation of the trace of Listing 7.2

```
...
<Trace abstraction>
  <InterpretatedQuery>
    Retrieve name of customer, number of order and date of order placed by
         customer living in Namur
    <AbstractedQuery>
      Select name, ncust, norder, date from customer, order where customer
          .ncust = order.ncust and locality = 'Namur';
    <AbstractedQuery>
  <InterpretatedQuery>
...
```

Listing 7.4 shows the conceptual interpretation resulting from the query interpretation level (see Section 5.6) of the trace described in Listing 7.2. Here, the loop was firstly abstracted as a join query (*Select customer.name, orders.norder, order.date from customer, order where customer.ncust = or-*

der.ncust and customer.locality = 'Namur'). Then, it was conceptually interpreted as *Retrieve name of customer, number of order and date of orders placed by customer living in Namur* based on the given conceptual schema.

7.5.2 Information Gathering

Listing 7.5: The XML structure of the prepared comments file

```
...
<comment location="8">Loop [8- 14]: customer.ncust <-> order.ncust</
    comment>
<comment location="8">Retrieve name, number of customer, number of order
    and date of order placed by customer living in Namur </comment>
...
```

Once we have selected the useful information to be injected as well as generated the appropriate comments, we can begin. This step seeks to identify the source code location where comments should be injected. To achieve this goal, we first need to recover the source code location of each SQL statement belonging to the analyzed trace. Then, we identify for each generated comment its corresponding location in the source code.

The above approach is actually independent of the technique chosen for the SQL location extraction. And we used the JDBC logger to identify the source code location of SQL statements. Here, we configured the JDBC logger to log the full stack trace that includes the following information: the executed queries, their results and their locations in the source code.

Once we have collected all the required information (comments and locations), the second step of this level involves associating comments with their corresponding source code locations in order to unify the input of the second level of the approach,

Listing 7.5 describes the XML structure of the gathered information through an example of the SQL query represented given in Listing 7.2.

7.6 Comments Injection

Starting from the XML file, this level seeks to inject comments into the program source code at their appropriate locations. For this, it needs an additional input, which is the program source code. Based on this input, this level provides as an output a commented source code.

To achieve these aims, we defined an algorithm that is described in Listing 7. The first step of the algorithm consists of extracting the AST of the program source code given as input (Line 1). Then, for each comment belonging to the XML file (Line 2), the algorithm traverses this AST in order to identify the location where the current comment should be injected (Line 3). After, the algorithm modify the AST by integrating the comment at the identified location (Line 5). Lastly, the algorithm applies all the changes made on the source code (Line 6).

Algorithm 7 Comments injection algorithm

```
Require: program source code ($SC$) and XML file of (comments/locations)
    ($XMLFile$)
Ensure: Re-documented program source code (with comments) ($RSC$)
 1: $ASTsc \leftarrow CreateAST(SC)$
 2: for all $elt \in XMLFile$ do
 3:     $position \leftarrow SearchPosition(elt.getLocation(), ASTsc)$
 4:     $comment \leftarrow elt.getComment()$
 5:     $ASTRewriteSC \leftarrow InjectComment(comment, position, ASTsc)$
 6:     $SCR \leftarrow ApplyChange(ASTRewriteSC)$
 7: end for
```

By applying the principle of the algorithm on the JAVA code of Listing 7.5.1.1 and the XML file containing the generated comments (Listing 7.5), we obtain the commented source code (Listing 7.6), where we have injected two comments (the first one describing the loop dependency between the two executed queries and the second one describing the conceptual interpretation of the procedural join which is implemented as a loop).

7.7 Discussions

The main contribution of this approach is the re-documentation of programs with its data-manipulation behavior by injecting comments about: (1) the conceptual interpretation of its SQL execution traces and (2) the dependencies that occur between its SQL queries. However, it is worth mentioning and discussing some untreated aspects as well as some borderline cases that may adversely affect the contribution of our approach:

- It should be noted that the analysis results that we obtained from the different levels of the approach and were described in Chapter 5 have not been properly scrutinized. More precisely, we assumed that all the results obtained were 100% accurate, which means that we neglected the possibility that results may include some noise. This is especially true for the algorithms on dependency detection. For instance, when the loops detection algorithm detects a false loop. In this case, the resulting abstracted trace, as well as the conceptual interpretation of

Listing 7.6: The commented program source code of the trace (Listing7.2)

```
...
public void SearchOrders(String nametab, String locality)
    throws SQLException{
...
this.connect();
String sqlselect="select ncust, name from "+nametab+" where
    Locality = ?";
java.sql.PreparedStatement preparedStatement1 =
    con.prepareStatement(sqlselect);
preparedStatement1.setString(1, locality);
//Loop [8-14]: costumer.ncust <-> order.ncust
//Retrieve name, number of customer, number of order and date of
    order placed by customer living in Namur
ResultSet res = preparedStatement1.executeQuery(sqlselect);
while (res.next()) {
String ncust = res.getString(1);
String sqlselect2 = "select norder, date from order where
    ncust=?";
java.sql.PreparedStatement preparedStatement2 =
    con.prepareStatement(sqlselect);
preparedStatement2.setString(1, ncust);
ResultSet res2 = preparedStatement2.executeQuery(sqlselect2);
...
}
...
}
...
```

the latter, will be insignificant. Thus, it is more judicious to reduce as much as possible noise to be able to retain only the most pertinent comments. To this end, analyzing the program source code may be a solution to validate the accuracy of the detected information. For instance, determining whether the detected potential loop is actually a loop in the program source code before injecting it corresponding comment.

- The generated comments are always injected just before query execution. However, we are not able to validate whether it is the most appropriate and useful location, where they should be injected. To this end, evaluating several possible source code locations in order to identify the most appropriate location may be an important aspect to take into account for the future improvements of this approach.

- In the proposed approach, we considered only results that are obtained from the analysis of a single SQL execution trace (the intra-scenario analysis 5). However, the results obtained from an analysis of multiple SQL executions traces to recover the data-manipulation process (Chapter 6) can also be useful for re-documenting program source code.

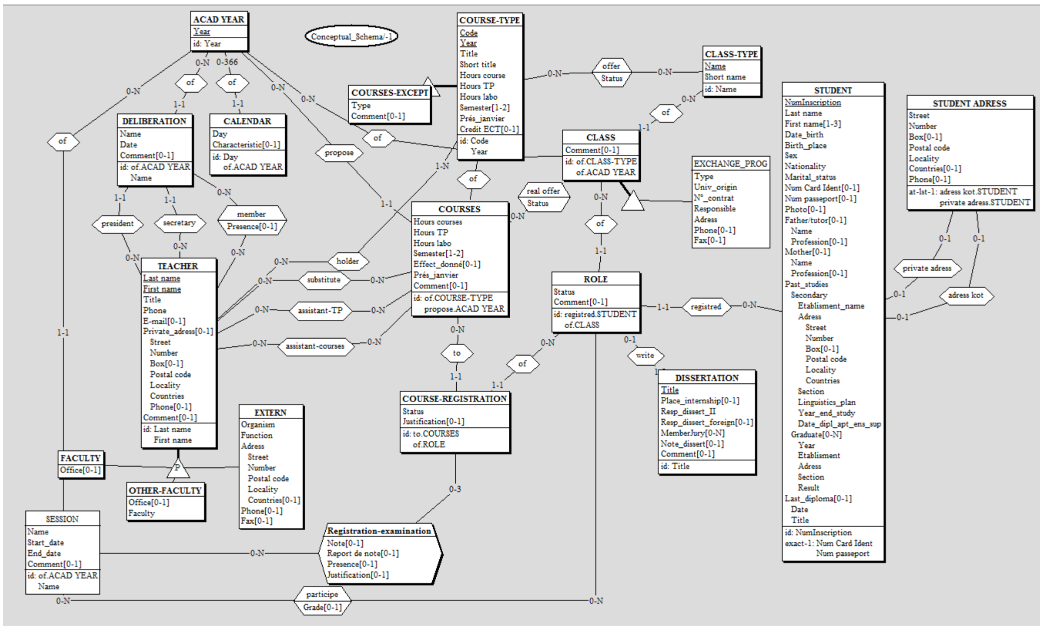


Figure 7.3: Conceptual schema of AcadYearManager application

7.8 Illustrative Example

Now, we present a example in which we apply our approach to demonstrate its feasibility and to show how it facilitates the understanding of program by adding comments to its source code.

To achieve this, we defined a Learning Management Systems (LMS) that allows the administration to manage all about teachers, teaching, students and courses, etc. This system, called AcadYearManager, consists of thousands of lines of code written in Java. It manipulates a MySQL database consisting of 30 tables and 192 columns representing the data on available

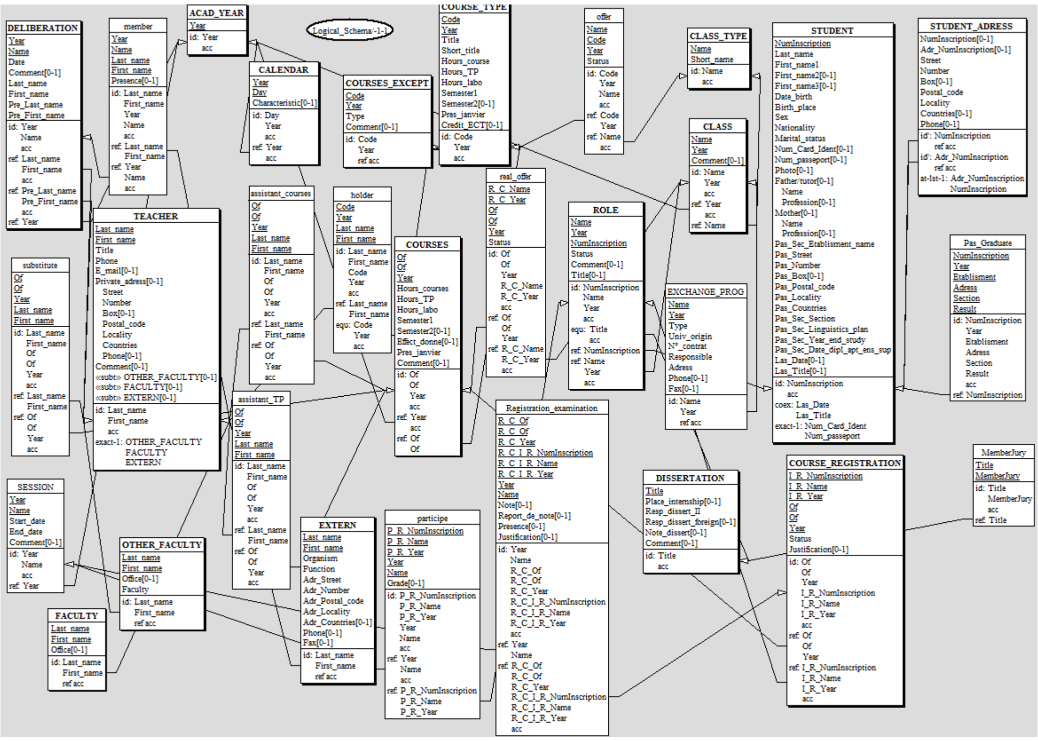


Figure 7.4: The logical schema of an AcadYearManager application

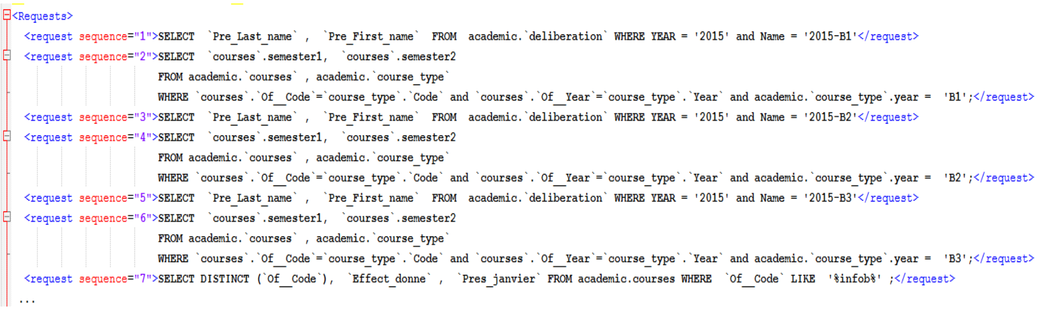

```
<Requests>
  <request sequence="1">SELECT  `Pre_Last_name` ,  `Pre_First_name`  FROM  academic.`deliberation` WHERE YEAR = '2015' and Name = '2015-B1'</request>
  <request sequence="2">SELECT  `courses`.semester1,  `courses`.semester2
                        FROM academic.`courses` , academic.`course_type`
                        WHERE `courses`.`Of__Code`=`course_type`.`Code` and `courses`.`Of__Year`=`course_type`.`Year` and academic.`course_type`.year =  'B1';</request>
  <request sequence="3">SELECT  `Pre_Last_name` ,  `Pre_First_name`  FROM  academic.`deliberation` WHERE YEAR = '2015' and Name = '2015-B2'</request>
  <request sequence="4">SELECT  `courses`.semester1,  `courses`.semester2
                        FROM academic.`courses` , academic.`course_type`
                        WHERE `courses`.`Of__Code`=`course_type`.`Code` and `courses`.`Of__Year`=`course_type`.`Year` and academic.`course_type`.year =  'B2';</request>
  <request sequence="5">SELECT  `Pre_Last_name` ,  `Pre_First_name`  FROM  academic.`deliberation` WHERE YEAR = '2015' and Name = '2015-B3'</request>
  <request sequence="6">SELECT  `courses`.semester1,  `courses`.semester2
                        FROM academic.`courses` , academic.`course_type`
                        WHERE `courses`.`Of__Code`=`course_type`.`Code` and `courses`.`Of__Year`=`course_type`.`Year` and academic.`course_type`.year =  'B3';</request>
  <request sequence="7">SELECT DISTINCT (`Of__Code`),  `Effect_donne` ,  `Pres_janvier` FROM academic.courses WHERE  `Of__Code` LIKE  '%infob%' ;</request>
 ...
```

Figure 7.5: A fragment of the collected SQL trace

courses, faculties, teachers, students, classes, etc. Figure 7.3 (resp. 7.4) depicts the conceptual schema (resp. the logical schema) of the AcadYear-Manager application.

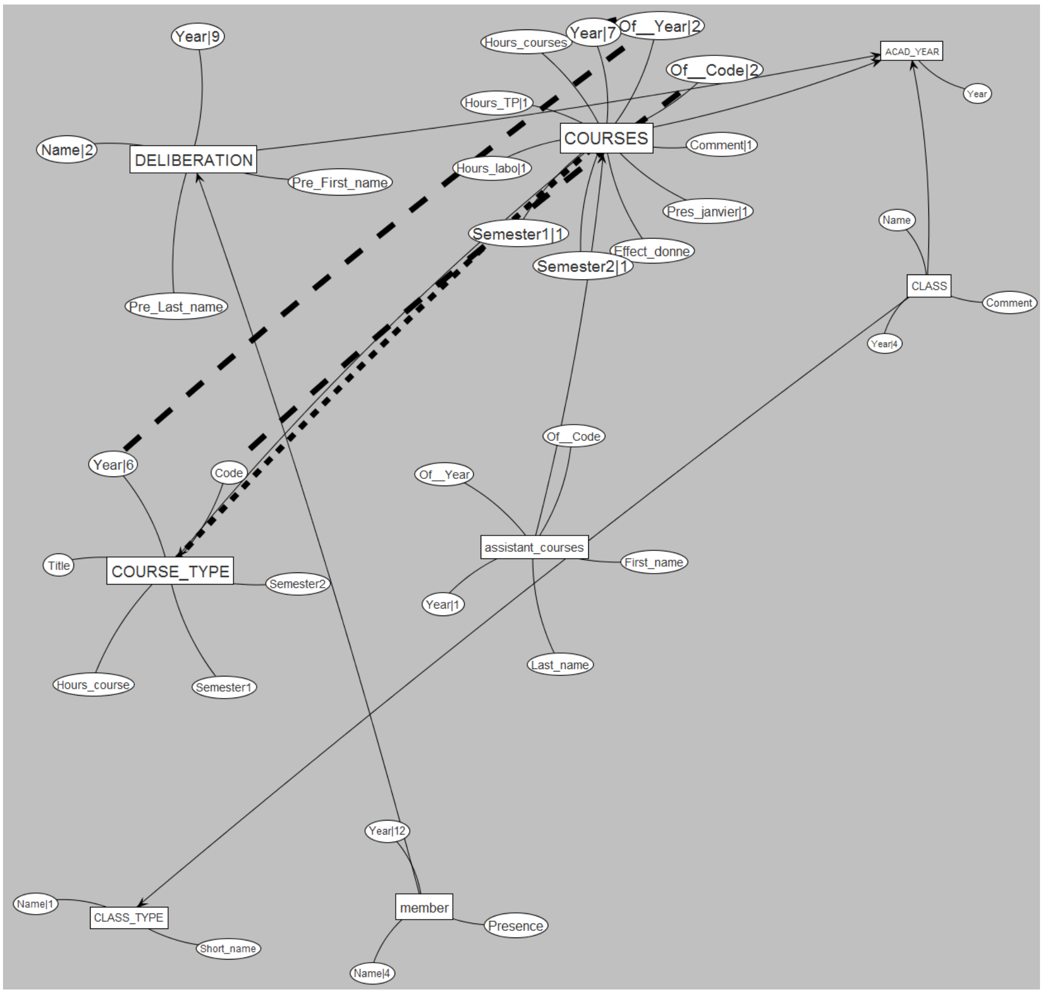
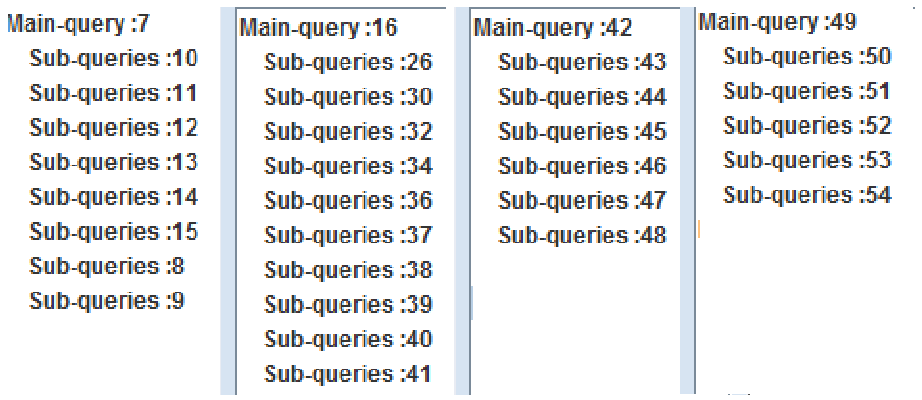


Figure 7.6: The intra-scenario analysis results

7.9 Results

In order to show the different steps of the approach, we collected an event log file with an SQL execution trace (SQL queries with their results) corresponding to one program execution scenario. The collected trace contains 69 queries (only select query). Figure 7.5 shows a fragment of the set of SQL queries belonging to the collected trace. After, we apply an intra-scenario analysis on this trace. Figure 7.6 depicts the analysis results, where it includes the subschema impacted by the «scenario 1» as well as the existing dependencies between queries belonging to the collected trace. And Figure 7.7 depicts the loops detection results. Here, the trace includes four possible loops, each of them being represented by the sequence number of its main



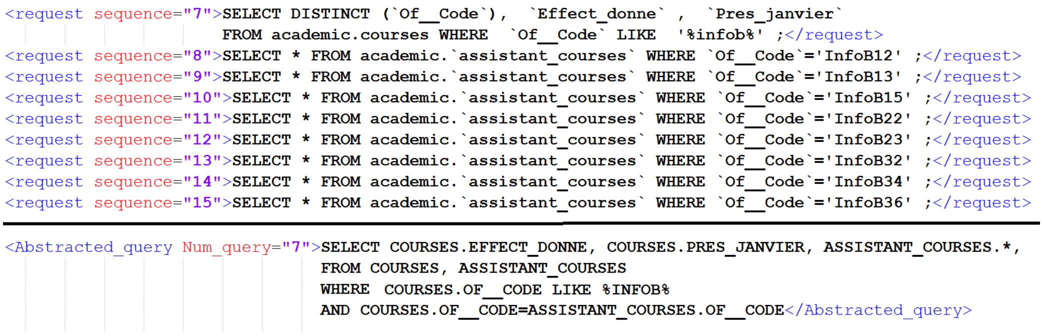
```
Main-query :7
    Sub-queries :10
    Sub-queries :11
    Sub-queries :12
    Sub-queries :13
    Sub-queries :14
    Sub-queries :15
    Sub-queries :8
    Sub-queries :9
```

```
Main-query :16
    Sub-queries :26
    Sub-queries :30
    Sub-queries :32
    Sub-queries :34
    Sub-queries :36
    Sub-queries :37
    Sub-queries :38
    Sub-queries :39
    Sub-queries :40
    Sub-queries :41
```

```
Main-query :42
    Sub-queries :43
    Sub-queries :44
    Sub-queries :45
    Sub-queries :46
    Sub-queries :47
    Sub-queries :48
```

```
Main-query :49
    Sub-queries :50
    Sub-queries :51
    Sub-queries :52
    Sub-queries :53
    Sub-queries :54
```

Figure 7.7: The loop detection results



```
<request sequence="7">SELECT DISTINCT (`Of__Code`),  `Effect_donne` ,  `Pres_janvier`
                      FROM academic.courses WHERE  `Of__Code` LIKE  '%infob%' ;</request>
<request sequence="8">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB12' ;</request>
<request sequence="9">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB13' ;</request>
<request sequence="10">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB15' ;</request>
<request sequence="11">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB22' ;</request>
<request sequence="12">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB23' ;</request>
<request sequence="13">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB32' ;</request>
<request sequence="14">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB34' ;</request>
<request sequence="15">SELECT * FROM academic.`assistant_courses` WHERE `Of__Code`='InfoB36' ;</request>
```

```
<Abstracted_query Num_query="7">SELECT COURSES.EFFECT_DONNE, COURSES.PRES_JANVIER, ASSISTANT_COURSES.*,
                                FROM COURSES, ASSISTANT_COURSES
                                WHERE  COURSES.OF__CODE LIKE %INFOB%
                                AND COURSES.OF__CODE=ASSISTANT_COURSES.OF__CODE</Abstracted_query>
```

Figure 7.8: The abstraction of the first detected loop

query as well as the sequence numbers of its nested queries.

Afterwards, the trace was abstracted by transforming the four procedural loops into join queries. As result, we got a new abstracted trace that contains 40 queries instead of 69. Figure 7.8 shows the abstracted query of the first detected loop (the top of the figure shows the initial trace, while the bottom of the figure shows the abstracted loop). Based on the annotated sub-conceptual schema, we obtained the conceptual interpretation of the abstract trace expressed in natural language terms. The abstracted query of the first detected loop of Figure 7.8 is interpreted as: *Retrieve all the characteristics of assistant_courses that give courses containing «info».*

7.10 Conclusions

The last phase of our global framework described in Chapter 4.4 is *Code re-documentation from SQL execution traces.* In this chapter, we proposed an approach for supporting program comprehension by re-documenting programs source code with their data-manipulation behavior. More precisely, we proposed a bottom-up approach that allows us to automatically generate: (1) comments about SQL statements dependencies (output/input and loops dependencies) and (2) a natural language interpretation of SQL statements.

In the next chapter, we will present an empirical study, which is a user experiment that attempts to assess the benefits of the approach implemented by DAViS.

Part III

Evaluation and Validation

Chapter 8

An Empirical Study on the Use of SQL Execution Traces for Program Comprehension

The only source of knowledge is experience.

Albert Einstein

Contents

8.1 Introduction 149

8.2 Related Work 150

8.3 Experimental Description 151

8.4 Experiment Execution 157

8.5 Results 159

8.6 Discussion 164

8.7 Threats to Validity 168

8.8 Conclusions 170

8.1 Introduction

In this chapter, we describe an empirical study carried out in an academic setting. It empirically assesses how developers/students are able to understand interactions between the database and the application program using DAViS, the tool supported by the proposed approach outlined in Chapter 5. To this end, we design a controlled experiment that quantitatively evaluates to what extent DAViS can influence program comprehension in terms of duration and correctness of the tasks. We also investigate which types of database manipulation comprehension tasks benefit most from the trace visualization support provided by DAViS. To address these questions, the experiment quantitatively measures how DAViS influences (1) the time required to achieve typical database manipulation comprehension tasks, and (2) the correctness of the answers related to these comprehension tasks.

The remainder of the chapter is organized as follows. Section 8.2 provides a summary of the related literature in the context of empirical studies. Section 8.3 gives a description of the experiment, while Section 8.4 describes the experimental setting. In Section 8.5, we summarize the results of our experiment, which are then discussed in Section 8.6. In Section 8.7, we elaborate on the threats to validity, then in Section 8.8 we make some remarks and draw some pertinent conclusions.

8.2 Related Work

Several related empirical studies [64, 90, 35, 14] inspired us to design our own experimental protocol.

In [64], the authors perform a controlled experiment in order to compare the development times using Java and Groovy with Eclipse IDE.

Wettel et al. [90] perform a large-scale evaluation of CodeCity [89], a 3D software visualization tool based on a city metaphor. The purpose of the controlled experiment was to provide empirical evidence that the tool had a positive impact on program comprehension tasks, taking IDE-based source code inspection as a baseline.

The controlled experiment described in [14] seeks to evaluate Extravis, a tool for the visualization of large execution traces. The main objective of the experiment was to measure to what extent Extravis could reduce the response time and the correctness of typical program understanding tasks, as compared to a simple source code inspection in an IDE. This experiment was similar to the one presented in this chapter. However, (1) we focus on understanding the interactions between the programs and its database, rather than understanding the interactions within and between source code entities; (2) we compare the performance of participants when trying to understand SQL execution traces (with *vs.* without our visualization tool), rather than when trying to understand the program's source code (with *vs.* without a visualization tool). In other words, in our experiment the input of the comprehension tasks is the SQL execution trace, not the corresponding source code fragment.

In [74], the authors present an empirical study which established that

Entity-Relationship (ER) models are easier to understand than their corresponding SQL data definition language (DDL) code. The study consisted of providing students with several ER models and some other students with their corresponding DDL code. They then compared their performance in answering questions about the databases represented by these artifacts. In contrast with our study, the authors focused on the understanding of SQL DDL statements (e.g., *create table* statements), which describe the *structure* of the database. Here, we focus on understanding SQL data-manipulation language (DML) queries (e.g., *select-from-where* statements) that allow programs to manipulate the database *contents*.

8.3 Experimental Description

8.3.1 Initial Considerations

The main purpose of the experiment is to quantitatively evaluate the effectiveness and efficiency of the approach implemented by DAViS – inspired by studies such as [93] and [16].

Our analysis of the SQL execution traces is the main contribution of the approach defined in Chapter 5. This provides users with a preliminary understanding of what happens in the program, in terms of participating objects and how they depend on each other. In this chapter, we analyze the possible impact of DAViS on comprehension for different tasks where each task has a specific purpose. For the visualization, we assume it allows a *quick* understanding. And finally, we assess whether different levels of education (Bachelor VS Masters students) get a different benefit from the tool. Thus, we want to measure the response time for each task. Furthermore, we consider non-trivial traces: in the experiment, we capture traces with an average size of 60 SQL queries. Hence, our goal is to design an experiment with the independent variable *tool* (factor with and without DAViS), the independent variable *task* (with different tasks to be achieved in the experiment), the independent variable *level of education* (factor with Bachelor and Masters), and the dependent variables *response time* and *correctness.*

In order to gain more precision and confirmation, it is necessary to collect participants' sentiments via a debriefing questionnaire about proposed tasks, complexity, the help proposed and the understanding gained.

8.3.2 Evaluation Questions and Hypotheses

The goal of the experiment is to reflect on three evaluation questions related to the use of DAViS:

- ***EQ***$\boldsymbol{EQ_1}$: *Does DAViS influence the time needed to complete the tasks?*
- $\boldsymbol{EQ_2}$: *Does the educational background influence the usability of DAViS?*
- $\boldsymbol{EQ_3}$: *Does DAViS influence the correctness of the tasks?*

In order to address these evaluation questions in an experiment, we will transform them into testable hypotheses:

- $\boldsymbol{H0_1}$: *There is no difference in the response time for different tasks between participants using DAViS and those that do not use DAViS.*
- $\boldsymbol{H0_2}$: *There is no difference in response times for Bachelor or Masters students.*
- $\boldsymbol{H0_3}$: *There is no difference in the correctness of responses between participants using DAViS and those that do not use DAViS.*

8.3.3 Used Scenario

We evaluate the approach in the context of AcadYearManager system – a Learning Management Systems (LMS) described in Section 7.8 that allows the administration to manage all the data about teachers, teaching, students and courses, etc. The first step that we performed on the AcadYearManager system was to collect SQL execution traces corresponding to four typical program execution scenarios. For each scenario, we recorded an event log file with the SQL queries executed during the scenario. For each executed query in the trace, we also recorded its result. Each collected execution trace contains a set of SQL queries, each belonging to one of the four different

Scenario	#Queries	Types of queries
Scenario 1	69	Select queries
Scenario 2	60	Select queries
Scenario 3	65	Select queries
Scenario 4	62	Select queries

Table 8.1: Characteristics of the four scenarios in the given system

scenarios. Table 8.1 summarizes the content of the four SQL execution traces, while Listing 8.1 shows four SQL queries belonging to the execution trace of *"scenario 2"*.

Listing 8.1: An example of the trace extracted from scenario 2

```
...
SELECT DISTINCT (Of_Code), Hours_courses FROM academic.courses;
SELECT * FROM academic.assistant_courses WHERE Of_Code="infob12";
SELECT Name FROM academic.member WHERE Name LIKE  % 2015%;
SELECT Name, Year FROM academic.exchange_prog;
...
```

8.3.4 Tasks Design

Question	Description
Q1	What are the tables which are accessed in the scenario?
Q2	What are the columns which are accessed in the scenario?
Q3	What are the objects (tables and columns) which are accessed the most frequently in the scenario? And how many times?
Q4	What are the pairs of tables which are accessed jointly in the scenario (within the same query)?
Q5	What are the input/input dependencies between the following pairs of tables?
Q6	How many queries would fail in the scenario, if we renamed this table?
Q7	How many queries would fail in the scenario, if we deleted this column?
Q8	What are the nested queries in the following set of queries?

Table 8.2: Comprehension tasks

We defined a set of eight comprehension tasks based on the work presented in [14]. The tasks are described in Table 8.2. Note that the tasks were chosen to highlight one or more aspects of the program understanding proposed by the different levels of understanding of the proposed approach.

The first level focuses on the usage statistics of each database schema element involved in the execution scenario (questions 1, 2 and 3). For instance,

Visualization	**Description**
Sub-schema	The sub-schema affected by the scenario
Frequencies	The frequency of each accessed element
Dependencies	The dependencies between consecutive elements
Nested queries	The nested queries in the scenario

Table 8.3: Visualization proposed by DAViS

question 1 seeks to identify all tables used in the input scenario. For this set of questions, DAViS provides a visualization of the subschema affected by the trace given as input. Then, at the second level, we will focus on the global behavior of the data-manipulation of the program (questions 4 to 8). This set of tasks attempts to focus on a slightly higher level of understanding, where we wish to understand the dependencies that exist between successive queries of the same scenario. In this way, we can in turn better understand the links that exist between distinct elements of the database schema. In this case, DAVIS provides a visualization for each kind of dependency. In Table 8.3, we summarize the different visualizations proposed by DAViS in the context of the evaluation.

Note that in our case, a task involves answering a question. Each task highlights one or more program comprehension aspects at a given level of understanding.

8.3.5 Pilot Studies

	Group 1	**Group 2**
Round 1	Scenario 1/DAViS	Scenario 1/~~DAViS~~
Round 2	Scenario 2/~~DAViS~~	Scenario 2/DAViS

Table 8.4: Experiment layout

Before defining the final experiment, we realized four pilot studies in order to optimize and adjust several experimental parameters, such as the number of questions, their order, their clarity, feasibility, the time limit and time taken to solve problems as they emerged in each pilot study. We performed the pilots studies in three steps:

1. The first pilot study for the experimental group was performed by *Ste-*

fan Hanenberg[1] an expert in empirical analysis with whom we collaborated to realize this experiment. Initially, it was not part of the design of the approach or the content of experimental design.

2. The second step sought to repeat the same pilot once the changes suggested by the previous one had been applied. This pilot had exactly the same objectives as the previous one. In addition, we wanted to estimate the time needed to finish the experiment. The pilot was conducted by a graduate student in his first year.

3. In the third step, we conducted two pilot studies, one for the control group and the second for the experimental group. The two pilot studies were conducted by two PhD students in their first year.

It is worth noting that the four pilot studies hadn't any information about the approach before the pilot. The results of these pilots studies led to a change in the question order and to combine two groups of questions related to the database schema elements involved, and to the dependencies between successive queries because some of the questions are related to both categories. It also allowed us to remove two questions about the feelings of subjects because they were too abstract and very difficult to evaluate and compare. Furthermore, the pilot studies led us to rephrase several questions and add some missing information in order to make the questions clearer.

8.3.6 Experiment Procedure

The experiment was designed as an 8x2 crossover trial where eight different tasks were performed by the participants with and without the DAViS output. The first group solved all the tasks first with the help of DAViS (while the second group solved it without the help of DAViS) and then solved the same tasks in a different scenario without the tool (the second group vice versa). Table 8.3.5 summarizes the experimental layout, while Figure 8.1 describes the handouts of each group.

The first independent variable is the availability of DAViS' visualizations during the experiment. This variable has two treatments (with and without the help of DAViS), while the second independent variable is tasks (the answer to each question), which has eight different treatments. The dependent variables in the experiment are the time needed for each question (measured in seconds), and the number of correct answers (measured in terms of being correct or incorrect).

[1] https://www.dawis.wiwi.uni-due.de/team/stefan-hanenberg/

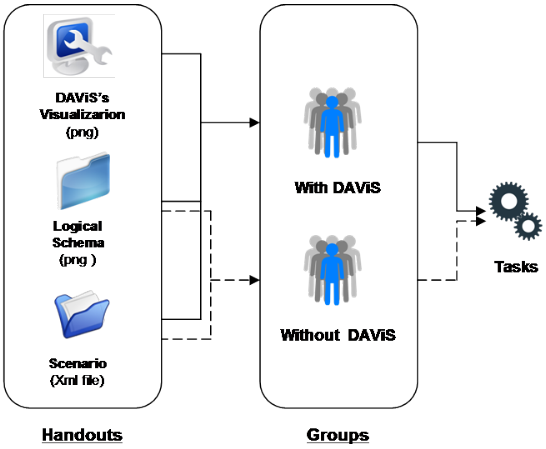


Figure 8.1: Handouts for each group

8.3.7 Data Collection

We created a MySQL database to record, during the experiment, all relevant information that could help us to measure the dependent variables. For the first dependent variable, we recorded the "start" time when the question was displayed to the participants and also the "end" time when the participants switched to the next question. Since going back to earlier questions was not allowed and the session was supervised, the time spent on each question could be easily found by determining the difference between the two times. For the second dependent variable, we recorded each time the participant attempted to provide an answer. This is time needed to calculate the number of trials before finding the right answer and then, going to the next question.

Additionally, we counted the number of skipped questions. It is worth mentioning that in order to move to the next question, it is either necessary to find the right answer or to exceed a period of 10 minutes without finding the correct answer. In this case, a skip button appears (after 10 minutes) in order to skip the current question and go to the next question. We added the option of skipping a question in order to keep participants in the experiment (when they were unable to answer a given question). At the same time, we were confident that the participants using DAViS would be able to answer the questions within 10 minutes (based on our experiences with the pilot studies).

Lastly, in order to get informal feedback from participants, we asked them to fill a debriefing questionnaire [2] at the end of the experiment. The participants were asked to assess the level of difficulty of each task and the

[2] Available at http://info.unamur.be/davis_debriefing

usefulness of visualization provided by DAViS, and (optionally) to share with us interesting insights about their experiences during the experiment.

8.4 Experiment Execution

8.4.1 Selection of Participants

After the pilot study involving four participants, we conducted the experiment with a total of 53 participants. The participants were chosen based on convenience: all participants were students at the University of Namur, either in their last year of the Bachelor or Masters studies. The students were a sample of all students attending the two programs. They were not selected based on their background, but simply because of their availability (with the requirement that they had some basic knowledge about SQL, browsing, database schema notations, DB-MAIN, XML, etc.). All the participants participated on a voluntary basis, and none of them had any prior experience of using DAViS. We randomly assigned the students to one of the two groups and this for each degree. In Table 8.5, we summarize the distribution of students in the two groups of each degree. It should be added that we didn't consider the results of 4 students. Because they went back to the previous questions, we couldn't compute the time spent on each question.

Students	**Groups**	
	Group 1	**Group 2**
Bachelor	16	14
Masters	9	10

Table 8.5: Distribution of students in the two groups

8.4.2 Concrete Setting

The experiment was performed at the university of Namur. It was conducted in two phases:

1. First, we commenced with a small presentation of an example that applied all the technologies used in the experiment, They were mostly the representation of the logical database schema using the DB-MAIN notation (e.g., tables, foreign keys, primary keys), and the XML file that contained a single SQL execution trace;

			Question 1		Question 2		Question 3		Question 4		Question 5		Question 6		Question 7		Question 8	
Grade	Group	Subjects	DAViS	~~DAViS~~	DAViS	~~DAViS~~	DAViS	~~DAViS~~	DAViS	~~DAViS~~	DAViS	~~DAViS~~	DAViS	~~DAViS~~	DAViS	~~DAViS~~	DAViS	~~DAViS~~
B	1	1	128	593	192	441	75	229	41	101	34	624	35	29	26	22	390	216
B	1	2	107	111	307	795	67	758	217	497	36	48	28	108	15	63	244	141
B	1	3	437	660	301	653	165	652	63	118	61	657	96	262	56	346	75	619
B	1	4	88	593	405	708	99	138	32	147	32	618	51	62	22	55	113	658
B	1	5	554	228	417	687	94	608	62	33	130	318	153	46	30	39	127	144
B	1	6	438	112	414	51	65	373	39	64	14	613	51	81	28	65	92	639
B	1	7	603	312	351	682	56	540	29	165	39	603	40	89	37	68	302	258
B	1	8	184	119	213	647	218	394	39	92	24	639	110	128	30	103	678	224
B	1	9	189	433	270	619	58	148	21	60	18	329	24	79	36	62	154	191
B	1	10	334	472	394	689	277	467	28	10	17	610	36	104	19	99	119	612
B	1	11	334	192	664	856	112	611	51	116	52	637	44	25	60	611	246	640
B	1	12	71	468	314	699	76	405	15	93	46	791	22	61	20	74	84	151
B	1	13	120	188	242	627	137	151	47	71	44	623	202	52	34	45	203	204
B	1	14	413	356	281	790	94	191	28	192	31	612	32	77	24	77	87	467
B	2	15	63	287	173	637	52	401	7	54	125	272	20	61	21	160	67	710
B	2	16	52	780	103	613	42	291	12	90	305	24	20	156	24	56	31	591
B	2	17	135	424	150	1020	60	611	7	70	96	547	12	30	11	41	121	614
B	2	18	96	347	253	637	55	138	8	155	388	27	21	435	16	160	40	606
B	2	19	53	175	208	969	63	204	10	111	85	554	74	36	19	25	46	318
B	2	20	39	267	154	1127	89	887	15	66	133	119	17	26	20	34	38	518
B	2	21	70	513	177	1046	60	232	10	57	347	100	24	28	29	52	96	635
B	2	22	56	242	216	1302	65	170	8	54	99	589	22	29	41	54	44	629
B	2	23	81	903	179	692	78	661	8	142	213	39	39	70	23	39	107	619
B	2	24	66	413	402	662	48	282	7	81	186	39	21	26	14	37	65	607
B	2	25	48	96	130	656	60	312	12	20	195	55	28	36	25	28	43	593
B	2	26	74	302	215	686	50	337	10	53	70	606	19	68	46	70	68	628
B	2	27	214	533	276	703	90	268	12	50	234	441	38	33	32	56	61	694
B	2	28	61	256	167	1025	49	356	10	23	185	145	22	37	17	38	137	610
B	2	29	64	781	215	674	42	222	8	79	78	118	16	33	20	29	45	622
B	2	30	125	515	162	934	50	216	9	214	138	134	44	77	24	40	41	627
M	1	31	77	175	190	391	112	151	20	39	15	617	29	62	17	66	66	628
M	1	32	112	189	223	587	92	157	27	72	37	636	56	159	23	113	74	616
M	1	33	105	525	211	624	63	686	18	147	27	639	29	34	14	36	44	630
M	1	34	165	140	493	809	91	650	58	88	20	608	187	118	98	166	80	607
M	1	35	165	160	248	660	84	443	55	68	35	506	31	43	42	58	75	669
M	1	36	168	175	252	678	83	393	68	45	28	641	39	33	18	111	65	622
M	1	37	122	373	266	562	87	621	59	124	126	627	129	54	26	64	76	466
M	1	38	169	173	253	383	131	151	27	85	53	261	49	51	23	109	69	702
M	1	39	259	729	298	620	105	443	35	217	22	640	35	207	35	172	184	653
M	1	40	319	421	251	710	105	348	45	192	44	629	50	49	36	79	152	634
M	2	41	33	152	107	675	32	98	13	110	211	156	28	63	28	39	47	612
M	2	42	83	480	166	1003	81	315	15	59	511	569	43	51	35	24	56	635
M	2	43	73	161	250	924	94	321	11	45	100	606	35	47	15	52	47	610
M	2	44	107	243	186	776	86	523	15	88	137	624	24	53	23	44	51	615
M	2	45	122	312	212	719	49	214	19	67	569	302	26	20	31	29	53	1829
M	2	46	79	182	200	668	71	263	47	62	134	215	38	41	18	42	68	957
M	2	47	87	206	167	772	51	269	23	154	118	265	33	33	30	44	51	552
M	2	48	61	984	236	2327	50	259	14	53	136	182	31	36	41	52	48	651
M	2	49	66	212	164	630	78	104	16	49	67	228	46	61	30	32	47	305

Table 8.6: Measured response times (in seconds)

2. Second, we conducted the test on workstations (one per participant). Each participant had to answer the questions of an on-line survey [3].

[3] Survey url: https://projects.info.unamur.be/eval_daviz/

The survey was mostly written in PHP and HTML/JavaScript. The debriefing questionnaire was prepared using Google Forms. All the participants had workstations with similar characteristics, i.e. a 17-inch screen with an identical screen resolution. Both sessions were supervised by at least two assistants to ensure the valid execution of the experiment. The experiment took about two hours in total, including the two rounds.

8.5 Results

The raw measurements of response times can be found in Table 8.6, which describes for each subject their level of education and the response times for each task (with or without DAViS).

The experiment was a three-factor crossover experiment that contained the following factors: education (Bachelor and Master), tooling (with and without DAViS), and tasks (with 8 different tasks). The crossover was performed on the factors tooling and tasks: after the first round, the subjects switch the tool (from with DAViS to without DAViS or vice versa) and redo the tasks for another scenario.

Due to the crossover design, there was the potential problem of a carry-over effect (such as learning effects when the subject switched from one tool to the other). To reduce the influence of this design choice on the results, we analyzed both rounds separately.

We ran a repeated measures ANOVA on the first round, as well as the partial eta-squared (η_p^2) to measure the effect size, we obtained the following results:

- **Within-subject factor task**: The factor tasks is significant ($p < .001$) with a large effect size ($\eta_p^2 = .534$).
- **Between-subject factor education**: The factor education is only approaching significance ($p = .061$) with a very small effect size ($\eta_p^2 = .076$).
- **Between-subject factor tool**: The between-subject factor tool is significant ($p < .001$) with a very large effect size ($\eta_p^2 = .774$).
- **Interaction effects**: There is no interaction effect between task and education ($p = .16$, $\eta_p^2 = .03$) and no interaction effect between the task, education, and the tool ($p = .49$, $\eta_p^2 = .02$). Furthermore, there

User: Namur
Password: PhD2018

is no interaction effect between education and the tool (p = .23, η_p^2 = .03). But there is a significant and medium interaction effect between the task and tool (p < .001, η_p^2 = .22).

Performing the same analysis on the second round gave the following results:

- **Within-subject factor tasks**: The factor tasks is significant (p < .001) with a large effect size (η_p^2 = .623).
- **Between-subject factor education**: The factor education is not significant (p = .441, η_p^2 = .013).
- **Between-subject factor tool**: The between-subject factor tool is significant (p < .001) with a very large effect size (η_p^2 = .854).
- **Interaction effects**: There are a interaction effect between the task and education (p = .026, η_p^2 = .32) and an interaction effects between the task, education, and tool (p = .006, η_p^2 = .37). And there is a very strong interaction between the task and tool (p < .001, η_p^2 = .90).

8.5.1 EQ1: Does DAViS reduce the time needed to complete the tasks?

8.5.1.1 Analysis of Response Times

Regardless of whether we consider the first round or the second round, the factor tooling is always significant and strong (with η_p^2 between .78 and .85). In terms of absolute values the differences in means with (M=120 s) and without DAViS (M=290 s) lay between 141 and 196 seconds (95% confidence interval) in the first round. In the second round the differences were higher: with (M=83 s) and without (M=340 s) DAViS led to a difference between 225 and 289 seconds (95% confidence interval). Hence, $\boldsymbol{H0_1}$ can be rejected.

Although this overall result is valid, it conceals the fact that the factor task itself is significant and strong (meaning that the response times among tasks are different – independent of the other factors) and that there was a significant interaction between the task and tool (where there is a difference in strength between the first and the second rounds). Furthermore, we should take into account the fact that the tasks were different: the first three tasks referred to the frequency of elements in queries, while the last five tasks referred to the dependencies of queries.

Q	p-value	Means (s)		95% conf. interval	Mean diff.	Saved time (%)
		DAViS	~~DAViS~~	DAViS -~~DAViS~~		
1	.012	236	391	273 ; 36	155	40
2	< .001	310	556	373 ; 119	246	44
3	< .001	106	318	289 ; 135	212	67
4	.009	47	80	58 ; 9	33	41
5	< .001	41	278	327 ; 147	237	85
6	.941	65	63	38 ; -41	-2	-3
7	.021	32	51	35 ; 3	19	37
8	<.001	158	590	513 ; 349	432	73

Table 8.7: T-tests on individual tasks (round 1)

8.5.1.2 Task-Wise Analysis on Response Times

While the (ordinal) interaction between the factors shows for both rounds that DAViS reduced the response times, the factor task is significant, i.e. the differences in response times depending on the kind of task. Hence, it is better to analyze the impact of the tool on each task separately.

Taking into account the fact that we did find any a significant effect of education, we ran each task on the whole sample (i.e. without distinguishing between levels of education) and the independent T-Test in the first round as well as in the second round.

Table 8.7 summarizes the t-test results (for round 1), where all tasks except task 6, reveal significant differences with p<.05. Furthermore, only tasks 1 and 7 display a p-value above .01, while all others are below .01. While Figure 8.2 shows the interaction diagram for the between-subject factor tool and the within-subject factor task: in all cases, the group using DAViS required less time than the group without DAViS except task 6.

Repeating the same analysis for the second round gave the results as shown in Table 8.8. In the second round, all tasks show a significant benefit for DAViS with p<.05. Furthermore, only task 7 has a p-value equal to .002, while all others are below .001. However, we see as well that there is a tendency that the differences in means are larger, an indication that there are some novelty effects (with the assumption that the learning effects are counterbalanced by the crossover). While Figure 8.3 shows the interaction diagram for the between-subject factor tool and the within-subject factor task: in all cases, the group using DAViS required less time than the group without DAViS with a large difference.

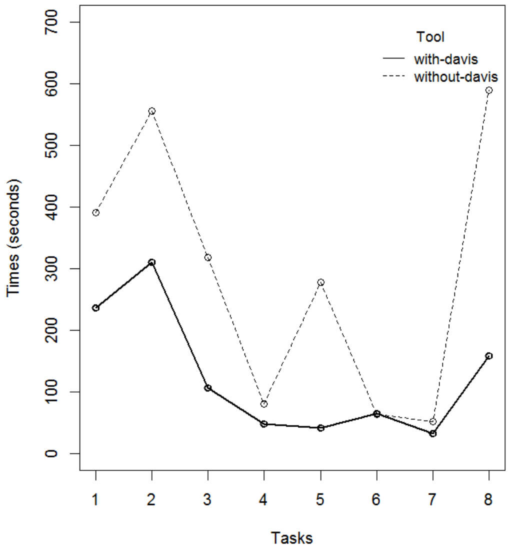


Figure 8.2: Interaction diagram of round 1 for the between-subject factor tool and the within-subject factor task

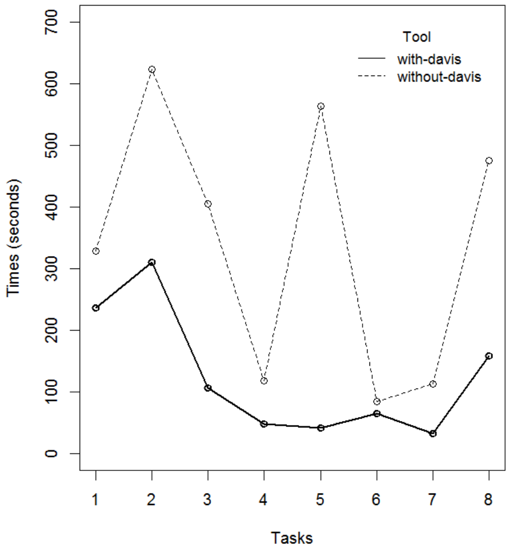


Figure 8.3: Interaction diagram of round 2 for the between-subject factor tool and the within-subject factor task

Q	p-value	Means (s)		95% conf. interval	Mean diff.	Saved time (%)
		DAViS	~~DAViS~~	DAViS -~~DAViS~~		
1	$< .001$	80	329	166; 331	249	76
2	$< .001$	195	624	354; 504	429	69
3	$< .001$	62	405	256; 429	343	85
4	$< .001$	13	118	64 ; 146	105	89
5	$< .001$	194	564	284; 455	370	66
6	$< .001$	30	84	29 ; 79	54	64
7	.002	25	113	35 ; 140	88	78
8	$< .001$	61	475	324; 504	414	87

Table 8.8: T-tests on individual tasks (round 2)

8.5.2 EQ2: Does the educational background influence the usability of DAViS?

In both rounds, the education factor was not significant, but in the first round it was approaching significance (with p=.061). However, it is essential to speak not only about the significance but also about the size of the effect. And in the first as well as in the second round the effect sizes were very small. Hence, although we should bear in mind that in the first round the results were approaching significance, we conclude (based first on the pure technical argument that the p-values were above .05 and that the sizes of the effect were below .1), hence we cannot reject $\boldsymbol{H0_2}$.

8.5.3 EQ3: Does DAViS increase the correctness of the tasks?

8.5.3.1 Task-wise Analysis of Correctness

In addition to the response time, we measured the correctness of the answers given by the subjects which is a dichotomous variable (i.e., we distinguish only between correct and incorrect). What we observed was that only two subjects were unable to give a correct answer to the question when the output of DAViS was available (subject 11 in task 2 and subject 17 in task 5). In all other cases, we got correct answers from the subjects for both rounds. For the non-assisted groups, the results were rather mixed: While for example most subjects were able to give a correct answer for question 6 (resp. question 7) without DAViS (with one exception for each) only 5 subjects were able to give a correct answer for question 2.

Round 1		Round 2	
Questions	p-value	Questions	p-value
1	.49	1	.11
2	$< .001$	2	$< .001$
3	.235	3	.002
4	$>.99$	4	constant
5	$< .001$	5	$< .001$
6	$>.99$	6	constant
7	constant	7	0.49
8	$< .001$	8	$< .001$

Table 8.9: Left: Exact Fisher tests on individual tasks about correctness (round 1), right: Exact Fisher tests on individual tasks about correctness (round 2)

We ran for each task on the whole sample (i.e. without distinguishing between levels of education) the exact Fisher tests in the first round as well as in the second round. Table 8.9 summarizes the results of both rounds separately. We get for the first round as well as for the second round significant results for tasks 2, 5, and 8 (each $p < .001$), where the group using DAViS had correct results significantly more often, while the other tasks had no significant differences. For task 3, we got significant differences in round 2 ($p < .05$), but not for round 1 ($p = .235$). Note that tasks 4 (round 2), 6 (round 2) and 7 (round 1) are *«constant»* which means that there is no variability deviation and then the p-value cannot be calculated, this is due to the fact that the two groups obtained exactly the same results.

Hence, in 3 of the 8 cases, we rejected hypothesis $\boldsymbol{H0_3}$, while in 5/8 cases we could not reject hypothesis $\boldsymbol{H0_3}$.

8.6 Discussion

In this section, we are going to discuss the results of the experiment. To this end, we also considered the outcome of the debriefing questionnaire, which is described in tables 8.10 and 8.11. Here, Table 8.10 summarizes the participants' difficulty in the perception of each task. From the table, we can clearly state that on average 90% of participants with the help of DAViS considered the tasks easy. And Table 8.11 summarizes the participants' perception of DAViS' help for each task.

Perceived task difficulty with DAViS (%)				
Task	**Trivial**	**Easy**	**difficult**	**Impossible**
T1	82.7	13.5	1.9	1.9
T2	63.5	28.8	3.8	3.8
T3	59.6	32.7	5.8	1.9
T4	67.3	23.1	9.6	0
T5	51.9	26.9	13.5	7.7
T6	61.5	32.7	5.8	0
T7	63.5	28.8	5.8	1.9
T8	59.8	25	0	13.5

Table 8.10: Debriefing questionnaire: perceived task difficulty with DAViS

Perceived help of DAViS (%)				
Task	**Yes, definitely**	**I think Yes**	**I think No**	**No, definitely**
T1	32.7	51.9	7.7	1.9
T2	30.8	57.7	3.8	1.9
T3	36.5	48.1	7.7	1.9
T4	30.8	48.1	15.4	0
T5	36.5	36.5	19.2	1.9
T6	25	40.4	23.1	5.8
T7	30.8	36.5	23.1	3.8
T8	36.5	38.5	7.7	11.5

Table 8.11: Debriefing questionnaire: perceived help of DAViS

8.6.1 Reasons for Different Time Requirements

The lower response time with DAViS-assisted subjects can be explained by the fact that all the information offered by DAViS provides a quick means to answer the questions. In others words, the proposed visualizations help one to answer a question. The time difference between the two rounds (tables 8.7 and 8.8) can be explained by a learning effect. In the second round, the participants repeated the same questions in the same order on the same system, but with a different scenario. We found that question 6 was not significant (we recall that question 6 concerns the number of queries that would fail if we renamed a table). This insignificant result could be explained by two reasons: (1) participants had already made the calculations for the table concerned in the previous tasks; (2) The search tool which was available to the participants (via the browser) allowed them to get exactly the right answer.

Still, several factors may have had an impact on the time requirements of DAViS users:

- ***The visualization proposed by DAViS:*** We consider the fact that all the information offered by DAViS's visualization makes it easier to answer a question. In addition, we proposed for each question the correct visualization, and this stopped the subjects from losing time by looking for the right visualization for the respective question. Based on the study on the visualization, we can explain this difference by the fact that visual information was much more accessible than textual information. In other words, retrieving information from a visualization is much faster than retrieving it from a text.

- ***Scenarios:*** We consider the size of traces (60 queries for scenario 1 and 65 queries for scenario 2) one of the major reasons for this time difference, where we had sufficiently large traces to be significant and reflect reality and small enough to be manipulable in an acceptable period of time. Still, two hypotheses may be deducted directly from the latter: (1) small traces will reduce the response time for the group not using DAViS and this is due to the fact that manipulation of the trace becomes much easier. More explicitly, if a small trace prevents users from scrolling the screen to find information, they will have all the information in a single view. (2) large traces will dramatically increase the response time of the groups who don't have DAViS's outputs. This causes a loss of concentration, and then the abandonment of the thread of the evaluation. In addition, we considered a non-trivial trace, and this may explain the time response difference. For instance, the search tool will not help users who are not using DAViS to directly find the right answer.

- ***DAViS's errors:*** In the experiment we conducted, we only reviewed the case where DAViS provided good answers, where the error rate is 0%. In this case, the DAViS-assisted subjects didn't care about the accuracy of the information provided by Davis. This may explain the big difference in time because otherwise (where visualization proposed by Davis may contain some errors) the DAViS-assisted subjects should check the accuracy of the information and this can take longer.

8.6.1.1 Individual Task Performance

We have seen the reasons that might influence the response time between DAViS-assisted subjects and the others in a general way. Now, we are going to discuss in more detail the time difference individually for each question. In this context, the results can be grouped into two large categories, which are supported by the debriefing questionnaire, where we can observe this categorization:

- ***Comparable results:*** Here, there was no large difference between the DAViS-assisted group and the non-assisted group. In this case, we have three questions (4, 6 and 7). Question 4 concerns the tables that are assessed jointly in the same query. For the DAViS-assisted subjects, this can be explained by the fact that DAViS provides a visualization highlighting all the joint tables, so they needn't take a long time to find the answer from the XML file. while the non-assisted subjects took more or less the same time, which is due to the complexity of the queries. More precisely, there were no sub-queries, which makes the research task easier. Then, participants could use the research tool to find the "From" clause where they could check easily if there is two tables. Questions 6 and 7 concern the identification of the number of queries that would be invalid if we delete (resp. modify) table (resp. column). The response time difference between the two groups (with and without DAViS) is not very large, which is due to the fact that the non-assisted group had already calculated the frequency of each affected object in questions 1 and 2. So they didn't need to recalculate it for the object concerned.

- ***Incomparable results:*** Here, there was a large difference between the DAViS-assisted group and the non-assisted group. This category includes questions (1, 2, 3, 5 and 8). Question 1 and 2 concern the identification of the tables (resp. columns) that are used in the scenario given as input. For the DAViS-assisted participants, this is due to the fact that DAViS provides the visualization of the subschema affected by the scenario given as input. And the non-assisted group had to read the XML file and get each object belonging to the scenario, which is a time-consuming task. This is also the case of question 3 and 4, where the question concerns the identification of the most affected table and column (resp. the identification of all input-input dependencies). In this case, the non-assisted group had to read the XML file as many times as the number of objects impacted by the scenario (resp. the number of possible combination). Lastly, question 8 seeks to identify all the nested queries. In this case DAViS provides the visualization in terms of boxes, where each box represents the main query and their nested queries, while the non-assisted group had to search and extract all the combinations from the XML file.

8.6.2 Reasons for The Education Level Independence

Regarding the education level of participants, we did not find any dependence between the level-study of participants/DAViS's added value. In other words, the participants didn't need to have a high level of expertise to use DAViS. Moreover, even with a strong background, DAViS brings an added value in response time and also in correctness. The result can be explained by the fact that the chosen comprehension tasks are not very hard to solve, but they are time-consuming. Thus, we may conclude that with or without some expertise, DAViS provides a considerable time-saving.

8.6.3 Reasons for Correctness Differences

We attribute the added value of DAViS in terms of correctness to two main factors. The first is the inherent precision of DAViS. The fact that DAViS only displays correct information allows one to increase the correctness for the DAViS-assisted groups. The second is that after several failed attempts to find the right answer, the non-assisted participants lose motivation and prefer to skip the task when it is possible to do so. For example, for task 2 (we recall that task 2 seeks to identify all columns used in the given scenario), only one participant from the DAViS non-assisted group was able to find the right answer. The other participants of the group have spent more than 10 minutes on it without finding the correct answer, and they eventually skipped the question.

8.7 Threats to Validity

This section describes the threats to our experiment's validity – something that is typically done in controlled experiments and explicitly asked for (see [35]). Here, we should reflect on possible influencing factors (that might or might not have appeared accidentally). It is worth noticing that there are different classification schemas for different types of threats of an experiment. We opted for the classification proposed by [12], and we are going to detail two categories, namely internal and external validity.

8.7.1 Internal Validity

- *Experiment Design:* We considered two different SQL executions scenarios, one in each round, where each scenario used more or less different objects in order to reduce the learning effect in the second round. However, it is possible that this goal was actually not completely achieved by the change of the given scenario because the types of questions and the subject system were the same in each round. Then there is a risk that a person's memory will affect the experimental results of the second round, as the participants know how the evaluation is conducted.

- *Participants:* In the context of this experiment, participants were not randomly chosen. Indeed, we conducted the experiment on students in their final year of their Bachelors or Masters degree, as part of a course on database modeling. However, the participation was not mandatory, it was just done on a voluntary basis. Therefore, we should note the following threats: (1) with student expertise, we should consider the threat that the students weren't sufficiently competent in the area of program comprehension. However, we made sure that they had basic knowledge about the concepts covered in the evaluation; (2) students motivation: knowing that the duration of the evaluation is about two hours, we may consider the threat of loss of motivation before and during the experiment, although the participation was on a voluntary basis.

- *Questions:* The definition of the questions may be influenced by the output proposed by DAViS. In addition, for each task, we proposed only the output that concerned the required information.

8.7.2 External Validity

The external validity refers to the conditions that limit our ability to generalize the results of the experiment to the realistic cases.

- *System/scenarios used:* The system on which we carried out the experiment may be viewed as a threat, because it does not attain the complexity of large systems used in the industry, in terms of numbers of tables in its database schema. Hence both the size and complexity of the scenarios were much smaller than the size of real-life execution traces.

- *Participants:* Another external validity threat is students. Undergraduate students may not be viewed as a fully representative population of the target profile of potential users of DAViS.

- *Questions:* Lastly, the representativeness of the questions is a potential threat to the validity of the results, i.e., the questions may not reflect real program comprehension task. It is possible that other kinds of tasks/questions matter more than the ones we considered.

8.8 Conclusions

In this chapter, we presented a controlled experiment seeking to evaluate how DAViS (a tool supported by the approach outlned in Chapter 5), can influence program comprehension in terms of duration and correctness of performing tasks. In the experiment presented in this chapter, we evaluated two components of DAViS, which are: (1) the intra-scenario analysis and (2) the inter-scenario analysis.

The results of the study indicate that DAViS does indeed reduce the response time and it increases the correctness (with a large effect size), which means that we found a strong indication that the chosen approach is truly able to help developers. This allowed us to answer our three evaluation questions and to validate our three hypotheses:

- The DAViS group needed 48% (on the scenario 1) and 76.75% (on the scenario 2) less time to achieve the selected understanding tasks.

- The added value of DAViS does not depend on the expertise-level of the participants. The gain in terms of response time remains significant, even when the participants have a higher level of expertise.

- The DAViS group achieved a higher level of correctness when answering the questions related to the understanding tasks.

This result is statistically significant, which means that there is a strong indication that the chosen dynamic analysis and visualization tool (DAViS) can greatly help developers. In order to determine which kinds of visualization are useful to gain a better understanding of the system, we analyzed separately the results of each user per question in more detail. In addition, we developed a debriefing questionnaire to identify the sentiments of the participants and verify our results.

Part IV

Conclusions

Chapter 9

Conclusions

Research is a journey, never a destination

Ralph Waldo Emerson

Contents

9.1 Summary of The Contributions 173

9.2 Future Challenges 179

The main objective of this thesis can be summarized by the following question:

How can we provide automated support for the program comprehension of data-intensive systems through an analysis of their data-manipulation behavior?

In the previous chapters, we presented in detail our framework, methodology and solutions to answer this question. In this chapter, we are going to summarize our contributions, draw some conclusions and suggest future challenges. We will answer our research questions and elaborate on the main lessons we have learned.

9.1 Summary of The Contributions

We viewed the problem of software understanding as an inevitable task for developers while adding a new feature, maintaining it or debugging an error. It is generally considered to be the most complex and costly task in terms of time and resources. Numerous approaches and tools have been proposed here to support the primary task of the software maintenance and evolution phase, which is program comprehension. However, modern data-intensive systems make the program understanding task even more difficult because of their dynamic and intensive use of data.

To overcome this problem, we first started by considering different fields relevant to this problem, namely, database engineering, database reverse engineering, program analysis, software visualization and process mining. Based on an in-depth study of these research fields (Chapter 2) as well as reviewing of the most relevant literature on what has been proposed in each of these domains (Chapter 3), we were able to: (1) first, highlight some problems and limitations of the existing solutions which have not been considered or treated so far; (2) second, formally define our problem statement and structure it through four main research questions.

Part 1: Understanding Data-manipulation Behavior from SQL Execution Traces		
Design	**Implementation**	**Evaluation**
C1: A trace parser	✓	- WebCampus - WebDeb - e-restaurant - AcadYearManager - Web store
C2: Subschema extraction algorithm	✓	- WebCampus - WebDeb - e-restaurant - AcadYearManager - Web store
C3: Dependencies extraction algorithms (O/I, I/I, joint access)	✓	- WebCampus - WebDeb - Web Store - AcadYearManager
C4: Loop detection algorithm	✓	- WebCampus - AcadYearManager
C5: Trace abstraction algorithm	✓	- AcadYearManager
C6: Conceptual interpretation algorithm	✓	- AcadYearManager - Illustrative examples
C7: DAViS – a visual tool supported by the approach	✓	- Empirical evaluation

Part 2: Extracting Data-manipulation Process from SQL Execution traces		
Design	**Implementation**	**Evaluation**
C8: Algorithms that extract, label and cluster data manipulation functionalities	✗	- Web Store

Part 3: Code Re-documentation Analysis		
Design	**Implementation**	**Evaluation**
C9: Algorithm generation comments	✓	- Illustrative examples
C10: Algorithm injection comments	✗	- Feasibility study

Table 9.1: Thesis contributions

To answer these questions, we proposed a framework comprising three incremental parts that support the program understanding of data-intensive systems via an analysis of their SQL execution traces (Chapter 4). This goal is achieved through a list of contributions distributed across these three parts. Tables 9.1 summarizes them according to three criteria; that is, the main objective of the contribution (design), whether the contribution has been implemented or not (implementation) and how was it evaluated (evaluation).

The remaining part of the contributions summary brings together the research contributions made while attempting to answer our research questions:

RQ 1: ***Can we automatically relate the program execution traces, program source code and the database schema to each other?***

In Chapter 4, we introduced a new framework to assist the program understanding of data-intensive systems based on an analysis of their SQL execution traces. The main idea is to dynamically analyze the SQL execution traces of data-intensive systems, i.e., interactions that occur between application programs and the database which are realized by SQL queries. Through an analysis of SQL execution traces, the framework was divided in three parts, where each part focused on: (1) understanding the data-manipulation behavior; (2) extracting the data-manipulation processes and (3) re-documenting the program source code through comments. The results obtained from the whole framework suggest that with the combined use of dynamic analysis, visualization and process mining, we can indeed relate the program execution traces, the program source code and the database schema.

RQ 2 : ***Does an analysis of the data manipulation behavior support the understanding of data-intensive programs?***

In Chapter 5, we addressed this question, by defining a bottom-up approach for supporting the understanding of data-manipulation behavior of programs from the analysis of a single SQL execution trace. The approach targeted in particular the analysis of data-intensive systems in the presence of automatically generated SQL queries and focused on relational databases. This approach identified four levels of understanding, relying on two inputs, namely a single SQL execution trace and database schemas (logical and conceptual). Here, a generic model (GER) was chosen as a pivot model for database schemas and CRUD operations as a SQL grammar model. The first level, called *the trace capturing level*, seeks to capture an SQL execution trace in order to parse it, so as to be able to extract the database objects used. The second level, called *the intra-query analysis*, seeks to highlight the sub-schema impacted by the trace given as input as well as the access frequency of each object used. The third one, called *the inter-query analysis*, seeks to extract dependencies (input-input dependencies, output-input dependencies and nested queries) between successive queries to understand how they depend on each other. And the last level, called *query interpretation*, seeks to interpret the SQL execution trace given as input from a more abstract point of view based on the conceptual schema.

In this chapter, we also presented a tool supported by the approach called DAViS (**D**ynamic **A**nalysis and **Vi**sualization of **S**QL execution traces). DAViS was implemented as a Plug-in to DBmain (a data-modeling and data-architecture tool). Based on the graphical representations of DBmain's schemas, DAViS provides a 2D visualization for each level associated with the approach. An initial experiment, based on two real-life applications (WebCampus and WebDeb), allowed us to establish that the analysis of SQL execution traces was a very promising technique for supporting the understanding of the data-manipulation behavior of data-intensive systems. As far as validation is concerned, in Chapter 8 we reported an empirical study for which the approach and DAViS presented in Chapter 5 had been empirically evaluated. The results were quite promising, which means that the suggested approach, and its supporting tool may support some program comprehension tasks.

RQ 3: ***How can we automatically extract a model of the data-manipulation behavior?***

In Chapter 6, we presented a bottom-up approach that sought to extract data-manipulation processes of data-intensive systems based on an analysis of SQL execution traces. The main idea is to dynamically analyze a multiple SQL execution trace of the same program execution scenario in order to recover the data-manipulation process followed by this one. The approach identifies two levels of understanding, relying on two inputs, these being a multiple SQL execution trace and database schemas (logical and conceptual). The first level, called *data-oriented properties*, first attempts to filter out from the given traces, all queries that do not belong to the conceptual schema of the system in question. Then, it associates with each SQL query, a set of properties that represents it. Afterwards, it clusters all queries that have the same set of properties, in order to associate with them a label that represents the data-manipulation function of the cluster. The second level, called *process mining*, seeks to recover the data-manipulation process followed by the program execution scenario which is represented by the SQL executions traces given as input.

In this chapter, we showed that using process-mining techniques may significantly contribute to the recovery of the data-manipulation process of programs, especially for the queries clustering and the identification of the variability of traces belonging to the same scenario. An initial experiment, based on an illustrative scenario and one small application (e-restaurant), gave promising results, which means that we found a strong indication that the selected approach can truly help developers to recover data-manipulation processes of data-intensive systems through an analysis of their SQL execution traces.

RQ 4: ***Can we automatically re-document this behavior in the program source code?***

In Chapter 7, we addressed this question by defining a new approach for the re-documentation of program source code via the injection of natural language comments expressing their data-manipulation behavior. Through these comments, the approach focuses on the frequent situations where such information is not available for the majority if not all current modern systems. This approach includes two steps, namely: (1) *comments generation*, which first attempts to identify what pieces of information should be considered. Then, according to the type of information, a natural language comment is generated; (2) *comments injection*, which makes a use of the AST representation of the program source code in order to inject the generated comments into the appropriate locations. Even though this chapter is at the exploratory stage, we did demonstrate its usefulness in the context of the program comprehension of data-intensive systems, as well as its feasibility through an illustrative example. The preliminary results indicate that this automatic mode of source code re-documentation is feasible and may assist program comprehension tasks.

We should also mention here that most of the contributions presented in this thesis were published at peer-reviewed international workshops and conferences, allowing us to in some ways validate our proposed approaches in the communities of experts in research field.

Links between Research Questions through Contributions

The ultimate goal of this thesis was to support the program comprehension task of the software maintenance and evolution phase. The dynamic

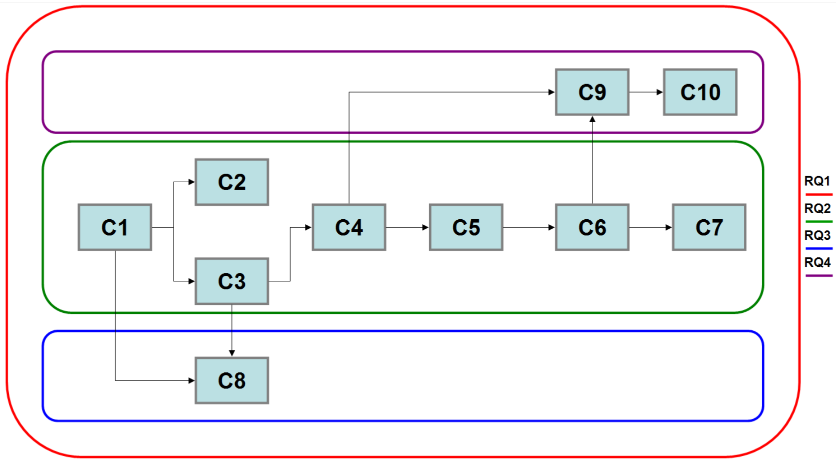


Figure 9.1: How the contributions are connected

analysis and visualization techniques support the understanding of the data-manipulation behavior of programs, by visualizing the analysis results of a single SQL execution trace. The process mining techniques support the recovery of data-manipulation processes, by analyzing a multiple SQL execution trace. The conceptual interpretation supports the automatic source code re-documentation through comments injection. Several important interconnections exist among the contributions (C) made with respect to the above research questions (RQ). We mean by an interconnection between two contributions (C1 and C2), the fact that the inputs of the contribution C2 depend on the outputs of the contribution C2. In other words, the contribution C2 is based on the results of C1. Figure 9.1 summarizes all the existing interconnections among the contributions by grouping them according to the research questions:

- *RQ2-RQ3, RQ2-RQ4*: the approach and the tool-supported defined in Chapter 5 seek to support the understanding of data-manipulation behavior through an in-depth analysis of a single SQL execution trace of a program. The chapter mainly focuses on the identification of the existing implicit dependencies between SQL queries, which in turn constitute the primary basis of the extraction of data-manipulation processes (Chapter 6) and source code re-documentation (Chapter 7).

- *RQ1-RQ2, RQ1-RQ3, RQ1-RQ4*: the global framework defined in Chapter 4 aims to support program understanding task, according to several levels of understanding divided into three parts (Chapter 5, Chapter 6 and Chapter 7).

9.2 Future Challenges

Our methodology has allowed us to achieve most of our goals in a core set of our defined problem. However, there remain some aspects that have not been considered and that may be extended in multiple ways but can also lead to further research topics. Indeed, this is due to the fact that we have made some assumptions and placed constraints in certain cases.

Here, we are going to suggest some possible improvements of the proposed framework that may overcome its existing limitations. Then, we propose some directions on how the tool we developed in the study could be extended and thoroughly validated. Lastly, we propose some future directions of research based on the results obtained in our study.

- **Enhancing of the defined framework:** The accuracy of the loop detection algorithm will be evaluated by the measures called precision and recall. In the experiment, we will perform a static analysis to check whether the detected loops really are loops in the program source code. Moreover, the use and usefulness of the conceptual interpretation of SQL execution traces will be evaluated through a user experiment. And the tool supported by the code re-documentation approach will be fully implemented and integrated with DAViS. Lastly, a systematic experiment could be conducted on the different implemented algorithms as well as DAViS, using real-world data-intensive systems.

- **Extending the scope of the inter-scenario analysis:** As outlined in Chapter 6, the main objective of the inter-scenario analysis was the extraction of the behavior of data-intensive systems in terms of a process model, e.g., a workflow describing with sequential, parallel and choice operators the sequence of execution of data-manipulation functions. However, we intend to consolidate and extend our initial approach with future prediction use. The main idea of this enhancement is to be able to calculate for each possible executed data-manipulation function, the probability of performing each possible function as the next one. In other words, we assign for each executed data-manipulation function the probability of moving to any other data-manipulation function. This analysis is based on the observed execution traces.

- **Extending the scope of dynamic analysis of SQL execution traces:** as discussed in chapters 5, 6 and 7, we just used the dynamic analysis of SQL execution queries in the context of data-intensive program comprehension. Hence, we intend to consolidate and extend our initial results, by exploring the use of dynamic analysis in domains other than data-intensive program comprehension, including a quality assessment for database queries, data security and consistency management.

Bibliography

[1] M. Alalfi, J.R. Cordy, , and T.R. Dean. WAFA: Fine-grained dynamic analysis of web applications. In *Proceeding of WSE'2009*, pages 41–50. IEEE CS, 2009.

[2] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[3] Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. Understanding web applications through dynamic analysis. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 120–129. IEEE, 2004.

[4] JoosC.A.M. Buijs, BoudewijnF. Dongen, and WilM.P. Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *OTM*, volume 7565 of *LNCS*, pages 305–322, 2012.

[5] P. Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, 2011.

[6] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. In *Readings in artificial intelligence and databases*, pages 98–111. Elsevier, 1988.

[7] A. Cleve and JL. Hainaut. Dynamic analysis of SQL statements for data-intensive applications reverse engineering. In *Proceeding of WCRE*, pages 192–196. IEEE CS, 2008.

[8] A. Cleve, N. Noughi, and JL. Hainaut. Dynamic program analysis for database reverse engineering. In *Generative and Transformational Techniques in Software Engineering*, 2012.

[9] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97:113–121, 2015.

[10] Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *Proceeding of the 13th Working Conference on Reverse Engineering (WCRE'2006)*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Anthony Cleve, Tom Mens, and Jean-Luc Hainaut. Data-intensive system evolution. *Computer*, 43(8):110–112, 2010.

[12] T.D. Cook and D.T. Campbell. Quasi-experimentation-design and analysis issues for field settings. In *Houghton Mifflin Compagny, Boston*, 1979.

[13] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, June 1989.

[14] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, May 2011.

[15] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. SE.*, 35(5):684–702, 2009.

[16] B. Cornelissen, A. Zaidman, A. van Deursen, and B. van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *Program Comprehension, IEEE International Conference on Program Comprehension '09.*, pages 100–109, May 2009.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[18] C. Del Grosso, M. Di Penta, and I. García Rodríguez de Guzmán. An approach for mining services in database oriented applications. In *Proceeding of European Conf. Software Maintenance and Reengineering 2007*, pages 287–296. IEEE Computer Society, 2007.

[19] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media, 2007.

[20] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-driven program understanding using concept analysis of execution traces. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 300–309. IEEE, 2001.

[21] E Allen Emerson and Edmund M Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *International Colloquium on Automata, Languages, and Programming*, pages 169–181. Springer, 1980.

[22] B. Ganter, R. Wille, and R. Wille. *Formal concept analysis.* Springer Berlin, 1999.

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

[24] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):425–456, 2006.

[25] Franklin C Grossman, David C Angel, and David A Seidel. Ir code instrumentation, November 16 1999. US Patent 5,987,249.

[26] M. Hack. *Petri Net Languages.* Computation Structures Group memo. Laboratory for Computer Science, 1976.

[27] Jean-Luc Hainaut. *Introduction to database reverse engineering.* Citeseer, 2002.

[28] Jean-luc Hainaut and Jl Hainaut. A generic entity-relationship model. In *in Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis, North-Holland.* Citeseer, 1989.

[29] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceeding of the IEEE 14th International Conference on Program Comprehension*, pages 181–190, 2006.

[30] Richard Healey, Steve Dowers, Bruce Gittings, and Mike J Mineter. *Parallel processing algorithms for GIS.* CRC Press, 1997.

[31] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[32] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Software Engineering, 1997*, pages 360–370, 1997.

[33] T. Kehl. The purpose of computing is insight, not numbers. *Transactions of the Society for Computer Simulation*, 7(6):280–280, 1966.

[34] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[35] H. Kitchenham, B.and Al-Khilidar, M. Babar, M. Berry, K. Cox, Jacky K., F. Kurniawati, M. Staples, H. Zhang, and L. Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*, 13(1):97–121, 2008.

[36] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

[37] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques.* Wiley Publishing, 1998.

[38] Thomas Kuhn and Olivier Thomann. Abstract syntax tree. *Eclipse Corner Articles*, 20, 2006.

[39] Y. Labiche, B. Kolbah, and H. Mehrfard. Combining static and dynamic analyses to reverse-engineer scenario diagrams. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 130–139, Sept 2013.

[40] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques, 2001.

[41] Boyang Li. Automatically documenting software artifacts. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 631–635. IEEE, 2016.

[42] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[43] Dien-Yen Lin and Iulian Neamtiu. Collateral evolution of applications and databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 31–40. ACM, 2009.

[44] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 270–281, New York, NY, USA, 2016. ACM.

[45] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. How do developers document database usages in source code? (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 36–41, Nov 2015.

[46] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011.

[47] Jonathan I. Maletic. An introduction to software visualization.

[48] Jonathan I Maletic, Andrian Marcus, and Michael L Collard. A task oriented view of software visualization. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 32–40. IEEE, 2002.

[49] R. S. Mans, Helen Schonenberg, Minseok Song, Wil M. P. van der Aalst, and Piet J. M. Bakker. Application of process mining in healthcare - a case study in a dutch hospital. In *BIOSTEC (Selected Papers)*, pages 425–438, 2008.

[50] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 262–273. IEEE, 2016.

[51] Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of dynamic database usage in java systems. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, pages 491–506, 2016.

[52] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963.

[53] Marco Mori, Nesrine Noughi, and Anthony Cleve. Extracting data manipulation processes from SQL execution traces. In *Information Systems Engineering in Complex Environments - CAiSE Forum 2014, Thessaloniki, Greece, June 16-20, 2014, Selected Extended Papers*, pages 85–101, 2014.

[54] Marco Mori, Nesrine Noughi, and Anthony Cleve. Mining SQL execution traces for data manipulation behavior recovery. In *Joint Proceedings of the CAiSE 2014 Forum and CAiSE 2014 Doctoral Consortium co-located with the 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014), Thessaloniki, Greece, June 18-20, 2014.*, pages 41–48, 2014.

[55] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[56] Csaba Nagy and Anthony Cleve. A static code smell detector for SQL queries embedded in java code. In *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*, pages 147–152, 2017.

[57] Csaba Nagy and Anthony Cleve. Sqlinspect: A static analyzer to inspect database usage in java applications. In *In ICSE '18 Companion: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden*, page 4 pages, 2018. to appear.

[58] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.

[59] Hamid R. Motahari Nezhad, Régis Saint-Paul, Fabio Casati, and Boualem Benatallah. Event correlation for process discovery from web service interaction logs. *VLDB J.*, 20(3):417–444, 2011.

[60] N. Noughi and A. Cleve. Conceptual interpretation of sql execution traces for program comprehension. In *In Proceeding PCODA*, pages 19–24, 2015.

[61] N. Noughi, M. Mori, L. Meurice, and A. Cleve. Understanding the database manipulation behavior of programs. In *Proceeding of IEEE International Conference on Program Comprehension*, page 4, 2014.

[62] Nesrine Noughi, Stefan Hanenberg, and Anthony Cleve. An empirical study on the usage of SQL execution traces for program comprehension. In *2017 IEEE International Conference on Software Quality, Reliability*

and Security Companion, QRS-C 2017, Prague, Czech Republic, July 25-29, 2017, pages 47–54, 2017.

[63] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, UK, 2002. Springer-Verlag.

[64] P. Petersen, S. Hanenberg, and R. Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *Proceeding of IEEE International Conference on Program Comprehension 2014*, pages 212–222. ACM, 2014.

[65] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[66] Jean-Marc Petit, Jacques Kouloumdjian, Jean-Francois Boulicaut, and Farouk Toumani. Using queries to improve database reverse engineering. In *Proceeding of the 13th International Conference on the Entity-Relationship Approach (ER'1994)*, pages 369–386. Springer-Verlag, 1994.

[67] Marian Petre, AF Blackwell, and TRG Green. Cognitive questions in software visualization, 1998.

[68] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[69] Philip J Pratt and Joseph J Adamski. *Concepts of database management*. Cengage Learning, 2011.

[70] Álvaro Rebuge and Diogo R. Ferreira. Business process analysis in healthcare environments: A methodology based on process mining. *Inf. Syst.*, 37(2):99–116, 2012.

[71] Gruia-Catalin Roman and Kenneth C. Cox. Program visualization: The art of mapping programs to pictures. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 412–420, New York, NY, USA, 1992. ACM.

[72] Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[73] Anne Rozinat, Ivo S. M. de Jong, Christian W. Günther, and Wil M. P. van der Aalst. Process mining applied to the test process of wafer scanners in asml. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 39(4):474–479, 2009.

[74] P. Sánchez, M. E. Zorrilla, R. Duque Medina, and A. Nieto-Reyes. Are models easier to understand than code? an empirical study on comprehension of entity-relationship (ER) models vs. structured query language (SQL) code. *Computer Science Education*, 21(4):343–362, 2011.

[75] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA, 2001.

[76] Mrinal Kanti Sarkar and Trijit Chaterjee. Reverse engineering: An analysis of dynamic behavior of object oriented programs by extracting uml interaction diagram. *International Journal of Computer Technology and Applications*, 4(3):378, 2013.

[77] Stephen R. Schach. Software engineering, fourth edition. In *Software Engineering, Fourth Edition, McGraw-Hill, Boston,*, page 11, 1999.

[78] John Stasko. *Software visualization: Programming as a multimedia experience.* MIT press, 1998.

[79] Tarja Systa. Understanding the behavior of java programs. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 214–223. IEEE, 2000.

[80] Richard N. Taylor. Assertions in programming languages. *SIGPLAN Not.*, 15(1):105–114, January 1980.

[81] Jonas Trümper, Stefan Voigt, and Jürgen Döllner. Maintenance of embedded systems: Supporting program comprehension using dynamic analysis. In *Software Engineering for Embedded Systems (SEES), 2012 2nd International Workshop on*, pages 58–64. IEEE, 2012.

[82] Huib van den Brink, Rob van der Leek, and Joost Visser. Quality assessment for embedded sql. In *Proceeding of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'2007)*, pages 163–170. IEEE Computer Society, 2007.

[83] Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves De Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose,

Peter van den Brand, Ronald Brandtjen, Joos Buijs, et al. Process mining manifesto. In *International Conference on Business Process Management*, pages 169–194. Springer, 2011.

[84] Wil MP Van Der Aalst and Schahram Dustdar. Process mining put into context. *IEEE Internet Computing*, 16(1):82–86, 2012.

[85] Jan Martijn EM van derWerf, Boudewijn F van Dongen, Cor AJ Hurkens, and Alexander Serebrenik. Process discovery using integer linear programming. *Fundamenta Informaticae*, 94(3):387–412, 2009.

[86] Boudewijn F Van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP Van Der Aalst. The prom framework: A new era in process mining tool support. In *ICATPN*, volume 3536, pages 444–454. Springer, 2005.

[87] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[88] R. Wettel and M. Lanza. Visualizing software systems as cities. In *IEEE Working Conference on Software Visualization' 2007*, pages 92–99, 2007.

[89] R. Wettel and M. Lanza. Visualizing software systems as cities. In *Proceeding of IEEE Working Conference on Software Visualization 2007)*, pages 92–99, 2007.

[90] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560. ACM, 2011.

[91] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 921–922, New York, NY, USA, 2008. ACM.

[92] David Willmor, Suzanne M. Embury, and Jianhua Shao. Program slicing in the presence of a database state. In *ICSM'2004: Proceeding of the 20th IEEE International Conference on Software Maintenance*, pages 448–452, Washington, DC, USA, 2004. IEEE Computer Society.

[93] C. Wohlin, M. Runeson, P., MC. Ohlsson, B. Regnell, A. Wesslén, and A. Von Mayrhauser. *Experimentation in software engineering : an introduction.* The Kluwer international series in software engineering. Kluwer Academic, Boston, London, 2000.

[94] Y. Yang, X. Peng, and W. Zhao. Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In *Proceeding of Working Conf. Reverse Engineering'2009*, pages 215–224. IEEE CS, 2009.

[95] WM Zuberek. Timed petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991.