Software Test Automation for Database Applications with Graphical User Interfaces

A dissertation

by

Haruto Tanno (丹野 治門)

Submitted to the

Department of Computer and Network Engineering

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the subject of

Engineering

The University of Electro-Communications

June 2020

Committee

Professor		Hideya Iwasaki
Professor		Yasushi Kuno
Professor		Masayuki Numao
Professor		Yasuhiro Minami
Associate	Professor	Minoru Terada

For refereeing this doctoral dissertation, entitled, Software Test Automation for Database Applications with Graphical User Interfaces.

Software Test Automation for Database Applications with Graphical User Interfaces

Haruto Tanno

Abstract

Software test automation plays a key role in improving the quality, cost, and delivery of software. The main tasks of software testing are test design, test execution, test results verification, fault localization, and program repair. Several tools automate these tasks in unit testing. Compared to integration testing, it is not difficult to remove bugs detected in unit testing.

Little headway has been made in automating integration testing, in which the tester checks the behavior of the software in combination with those of various modules such as user interfaces, application logic modules, and databases (DBs). Although the usage of automatic test execution tools such as Selenium has become widespread, test design and test results verification are often performed manually. Moreover, the functions tested in integration testing are affected by the operations of multiple modules, and their behaviors greatly depend on the input events and the program's internal states. This means that much time is required to identify the cause(s) of each bug detected in integration testing.

This study targeted DB applications with graphical user interfaces (GUIs) such as web and game applications. Test automation is indispensable for developing such applications because they evolve quickly and thus need early and frequent releases. We propose using four methods for solving the major problems we identified in test design, test results verification, and fault localization in functional testing at the integration level for DB applications with GUIs. Two methods based on model-based testing automatically generate initial DB states to be entered into the relational DB to support each test case. A third method, based on visual regression testing, enables the tester to efficiently confirm test results even when there are changes that affect the entire application screen. The fourth method, named suspend-less debugging, enables the programmer to efficiently debug interactive and/or realtime programs in the DB application with the GUI.

The goals of this study were to automate the process from test design to test results verification and to make it easier to identify the cause(s) of each bug detected and fix them.

Acknowledgments

Many people helped me complete my doctoral research. I would like to thank them here.

First of all, I would like to thank my supervisor, Professor Hideya Iwasaki for the continuous support of my study. His great guidance to me was essential for my research activities.

I am deeply grateful to the co-authors of the publications in my doctoral research, Katsuyuki Natsukawa, Yu Adachi, Yu Yoshimura of Nippon Telegraph and Telephone Corporation (NTT), Takashi Hoshino of NTT TechnoCross Corporation, and Xiaojing Zhang of NTT Communications. They spent a lot of their time discussing this research and gave me deep insights.

Advice and comments given by Professor Yasushi Kuno, Professor Masayuki Numao, Professor Yasuhiro Minami, and Associate Professor Minoru Terada have been a great help to improve this doctoral dissertation. I would like to show my greatest appreciation to them.

I owe a very important debt to members of NTT and NTT Group companies for supporting this research. In addition, I would like to offer my special thanks to members of Iwasaki laboratory.

Finally, thanks to my family for always warmly supporting my life and research.

Contents

Acknowledgments **i**

1 Introduction **1**
1.1 Background 1
1.2 Goals and Contributions 3

2 Testing of DB Applications with GUIs **7**
2.1 Database Applications with GUIs 7
2.2 Current Status and Testing Challenges 8
2.2.1 Test Design 9
2.2.2 Test Execution 11
2.2.3 Test Results Verification 11
2.2.4 Fault Localization 12
2.2.5 Program Repair 13

3 Design-Model-based Initial DB State Generation **15**
3.1 Introduction 15
3.2 Related Work 17
3.3 Proposed Method 18
3.3.1 Design Model 19
3.3.2 Test Case 22
3.3.3 Algorithm of Generating Initial Database State and Input Values 23
3.4 Evaluation 27
3.4.1 Measure 27
3.4.2 Procedures 28
3.4.3 Result 29
3.4.4 Discussion 30
3.5 Chapter Summary 32

4 Reducing Number and Size of Initial DB State **33**
4.1 Introduction . . . 33
4.2 Related Work . . . 36
4.3 Proposed Method . . . 38
4.3.1 Challenges . . . 38
4.3.2 Overview of Proposed Method . . . 39
4.3.3 Step 2: Test Case Grouping . . . 41
4.3.4 Step 4: Record Alignment Decision . . . 42
4.3.5 Step 5: Initial Database State Constraint Generation . . . 44
4.4 Evaluation . . . 46
4.4.1 Measure . . . 46
4.4.2 Procedure . . . 47
4.4.3 Result . . . 48
4.4.4 Discussion . . . 48
4.4.5 Future Work . . . 52
4.5 Chapter Summary . . . 52

5 Region-based Essential Differences Detection **53**
5.1 Introduction . . . 53
5.2 Related work . . . 55
5.2.1 Implementation-dependent VRT Systems . . . 56
5.2.2 Implementation-independent VRT Systems . . . 57
5.3 Proposed method . . . 58
5.3.1 Scope and Requirements . . . 58
5.3.2 Features of ReBDiff . . . 58
5.3.3 Difference Types . . . 59
5.3.4 Difference Checking by Tester . . . 61
5.4 Differences Detector . . . 63
5.4.1 Step 1: Divide Images into Regions . . . 64
5.4.2 Step 2: Generating List of Region Pairs . . . 66
5.4.3 Step 3: Assigning Difference Types to Region Pairs . . . 67
5.5 Experiments . . . 68
5.5.1 Research Questions . . . 68
5.5.2 Method . . . 68
5.5.3 Target Screens . . . 69
5.5.4 Results . . . 71
5.5.5 Discussion . . . 73

5.6 Conclusion 75

6 Suspend-less Debugging **77**
6.1 Introduction 77
6.2 Related work 80
6.3 Proposed debugging method 83
6.3.1 Targets 83
6.3.2 Requirements 83
6.3.3 Debugging method 84
6.4 Debugger for C# 85
6.5 Implementation 88
6.5.1 Mechanism of code transformation 89
6.5.2 Mechanism of debugging execution 90
6.6 Case study 91
6.6.1 Debugging First Bug 92
6.6.2 Debugging Second Bug 94
6.7 Overhead measurement 95
6.8 Discussion 96
6.9 Chapter Summary 97

7 Toward More Efficient Testing **99**
7.1 Reducing Cost for Design Model Creation 99
7.2 Expanding Scope of Initial Database State Generation 99
7.3 Reduction of Labor for Test Results Verification 100
7.4 Further Debugging Support 100

8 Conclusion **103**

Bibliography **105**

Publication List **112**

List of Figures

1.1 Tasks in Testing Process 1

2.1 Typical Execution Flow of Database Applications with Graphical User Interface 8
2.2 Integration Testing of Database Applications with Graphical User Interfaces 9

3.1 Example of Database Application 16
3.2 Overview of Proposed Method 19
3.3 Design Model: Process Flow of Employee Search System 20
3.4 Design Model: Input Definition of Employee Search System 20
3.5 Design Model: DB Schema of Employee Search System 21
3.6 Test Case of Employee Search System 23
3.7 Overview of Algorithm that Generates Initial Database State and Input Values 24
3.8 Generate Constraints from Containment Relationship of Strings 27
3.9 Decide Values of Character Variables 28

4.1 Example Test Cases 35
4.2 Initial Database States Generation by Existing Methods 36
4.3 Initial Database States Generation by Proposed Method 37
4.4 Problems When Naively Combining Records 39
4.5 Overview of Proposed Method 40
4.6 Test Cases Grouping 42
4.7 Example of Design Model 44
4.8 Examples of Record Alignment Tables 45
4.9 Example of Using Minimum Records Pattern to Obtain Solution 50
4.10 Example of Using Maximum Records Pattern Instead of Minimum Pattern to Obtain Solution 51
4.11 Example of Inability to Obtain Solution with Either Minimum or Maximum Records Pattern 51

5.1 Using Existing Image-based VRT to Compare Two Screens in Pixel Units. 54
5.2 Problems in Comparing Two Screens in Pixel Units with Existing Image-based VRT. 55
5.3 Overview of ReBDiff. 59
5.4 Detecting Differences in Region Pairs. 61
5.5 Shift-Checking View (Group Identifier is 1). 62
5.6 Addition/Deletion/Alteration-Checking View. 63
5.7 Checking for Alteration Differences. 64
5.8 Difference Detection Rates and Confirmation Times for RQ2. 72

6.1 Code Fragment for Action Game Program. 78
6.2 Views of SLDSharp. 84
6.3 Highlighting the Executed Statements. 87
6.4 Overview of SLDSharp Implementation. 88
6.5 Transformed Code. 89
6.6 Pseudo Code for `LogFunc`. 91
6.7 Mechanism of Debugging Execution. 92
6.8 Debugging First Bug. 93
6.9 Debugging Second Bug. 95

List of Tables

3.1 Definition of Design Model 21
3.2 Definition of Constraints on Where Clauses and Guard Conditions 22
3.3 Test Cases and Variables 23
3.4 Category of Test Case 29
3.5 Evaluation Results 29
3.6 Use Frequencies of Characteristics Constraints We Propose 29

4.1 Definition of Test Case Group 40
4.2 Calculate the Number of Records From the Result condition 42
4.3 Evaluation Result: Number of Initial DB States and Total Number of DB Records 48
4.4 Evaluation Result: Number of Backtrackings 49
4.5 Evaluation Result: Shuffled Test Cases 49

5.1 Target Screens. 69
5.2 Results for RQ1: Number of Detected Differences. 70
5.3 Results for RQ1: Ratio of Area. 71

6.1 Classification of Bugs and Scope of This Work 80
6.2 Requirements for Debuggers Applied to Interactive and/or Realtime Program. 82
6.3 Commands for Specifying Sections, Conditions, and Expressions to Monitor. 87
6.4 Overheads for SLDSharp. 96

Chapter 1

Introduction

1.1 Background

The objectives of software testing (hereafter "testing") are to verify that the target software has been implemented in accordance with its design and specifications and to reduce the number of software defects. The testing workload accounts for a large percentage of the overall development process. Moreover, since bugs that are not detected in the test process may affect the end users, the test process is vital to ensuring the quality of the software. User needs and software/hardware development of the operating environment have been evolving at an ever more rapid pace in recent years, meaning that early and frequent releases of new or revised versions of the software are needed. Maintaining software quality through repeated release cycles requires the performance of regression testing.

The testing process mainly consists of eight tasks: test planning, test design, test execution, test results verification, fault localization, program repair, test reporting, and test management, as shown in Figure 1.1.

Test planning In test planning, issues such as the time frame and allocation of resources for testing are decided on the basis of the overall development plan.

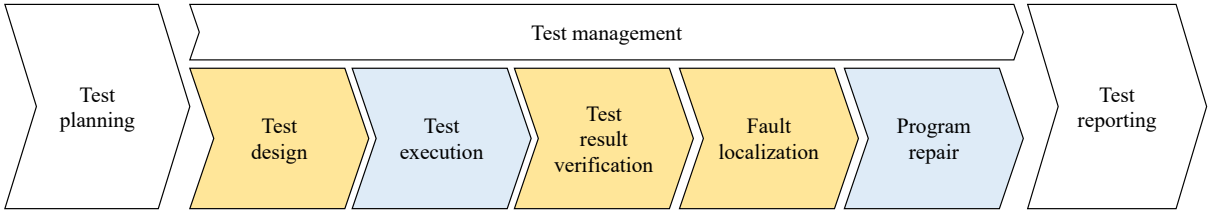


Figure 1.1: Tasks in Testing Process

Test design Test design involves clarifying the various tests that must be done, designing comprehensive test cases, refining specific test feasibility procedures for each test case, and creating scripts for automatic execution.

Test execution In test execution, test data are input for each test case one-by-one, the software is run, and the test results revealing how the software behaved for each test case are recorded.

Test results verification Test execution results are then compared with expected results to ensure that the software behaved in accordance with its design.

Fault localization If a behavior of the software is not as expected, that is, if a bug is detected, the program is corrected by identifying the cause(s) of the bug.

Program repair In program repair, the cause(s) of the bug is/are carefully removed while ensuring that the repair does not adversely affect the program.

Test management In test management, the progress of the tests is managed, and the test plan is revised if necessary.

Test reporting When all tests have been executed and all detected bugs have been fixed, the test results are summarized in test reporting, and the test process is complete.

Five of these test processes, namely test design, test execution, test results verification, fault localization, and program repair, are especially important. Once test cases are produced in test design, they can be repeatedly used not only for new testing but also for regression testing when software is fixed or improved. Test execution and test results verification must be carried out repeatedly for all legacy functions when dealing with software enhancements and new operating environments, and the burden increases exponentially as the scale of the software increases. Of course, the cost of debugging also increases with the scale and complexity of the software.

For unit testing, in which individual units of the source code are tested, tools such as Microsoft Visual Studio IntelliTest [76], which automatically performs test design, and JUnit, which automates test execution and test results confirmation, are used at development sites. It is relatively easy to identify the cause of a bug and fix it in unit testing since the functions at the unit level are relatively simple. In contrast, little headway has been made in automating integration testing, in which the tester checks the behavior of the software in combination with those of various modules, including user interfaces, application logic modules, and databases (DBs). Although automatic test execution tools such as Selenium[1] have become widely used, test design and test results verification are often performed manually. Moreover, a function tested in integration

[1] `https://selenium.dev/`

testing works in combination with the functions of multiple modules, and its behaviors greatly depend on the input events and the program's internal states. Therefore, it takes longer to identify the cause(s) of each bug detected in integration testing, and fixing the bug costs more due to the need to ensure that the modification does not adversely affect the program than to modify the program. The automatic performance of regression testing would enable a buggy program to be repaired relatively easily and safely. However, creating automated regression tests is costly.

1.2 Goals and Contributions

This study targeted DB applications with graphical user interfaces (GUIs) (hereinafter referred to as *db-gui-apps*) such as web and game applications because these types of applications have two characteristics in particular.

- There is a great need for test efficiency because many companies have developed db-gui-apps [1].
- They evolve quickly and need early and frequent releases, making test automation indispensable.

We propose using four methods to solve the major problems we identified in test design, test results verification, and fault localization in functional testing at the integration level for db-gui-apps. More specifically, the scope of our testing is limited to one screen transition for each test case, i.e., first tests in integration testing.

The goals of this study were to automate the process from test design to test results verification and to enable the cause(s) of each bug detected to be easily identified and removed.

The contributions of this dissertation are as follows.

- We discuss the current status and challenges of testing db-gui-apps and specify four major challenges faced in achieving the goals of this study.
- We propose using two methods based on model-based testing (MBT) that automatically generate initial DB states to be entered into the relational DB to support each test case. (Chapters 3 and 4))
- We propose using a method based on visual regression testing (VRT) that enables the tester to confirm test results efficiently even when there are changes that affect the entire application screen. (Chapter 5)

- We propose using a method named suspend-less debugging that enables the programmer to efficiently debug interactive and/or realtime programs in the db-gui-app. (Chapter 6)

- We discuss the remaining challenges to making db-gui-app testing more efficient. (Chapter 8)

As mentioned above, db-gui-apps evolve quickly and need early and frequent releases. In addition, there are various technologies for implementing db-gui-apps. Considering these points, we adopted two policies in the development of our four proposed methods. The first policy is that they be applicable even when the software functions have evolved. The second policy is that they be independent of the target software's implementation technologies.

For test design, we propose using a method called DDBGen for creating initial DB states and input values for each test case from the design model for industry-level enterprise systems. Our study identified the constraints most frequently used in industrial-level enterprise systems; they include "multiple DB reads," "PK, FK constraints," and "partial string matching." Since our design model can satisfy these constraints, high initial DB generation rates were achieved in evaluations on three industrial-level enterprise systems. In addition, we propose using a method called DDBGenMT for generating initial DB states in which each state is shared by multiple test cases to reduce the number of times the initial DB state must be switched and to reduce the total size of the test data.

For test results verification, we propose using an image-based VRT system called ReBDiff. It divides each of the images of the two application screens to be compared into multiple regions, makes appropriate matchings between corresponding regions in the two images, and detects differences on the basis of the matchings. By using ReBDiff, the tester can efficiently identify essential differences between the two screens even when there are changes that affect the entire screen, e.g., parallel movements of screen elements. Experiments using screens for PC web and mobile web services and an Electron application demonstrated the effectiveness of the proposed method.

For fault localization, we propose using a suspend-less debugging method for debugging logical errors in interactive and/or realtime programs. It displays information on execution paths and the values of expressions in a debuggee program in real time without suspending program execution. We implemented this method in SLDSharp, a debugger for C# programs. We demonstrated its effectiveness through a case study using a game program developed using the Unity game engine. The proposed debugging method was shown to enable a programmer to efficiently debug interactive and/or realtime programs. It is particularly suitable for debugging an application logic module that always produces

the same result for the same input. Therefore, SLDSharp is also effective for debugging web applications that always transition to the same screen for the same input. The method is generally applicable to many languages and target application domains.

Chapter 2

Testing of DB Applications with GUIs

In this chapter, first we describe db-gui-apps. Next, we discuss the current status and existing methods for testing db-gui-apps. Finally, we describe the four major challenges we tackled to achieve the goals of this study.

2.1 Database Applications with GUIs

A typical execution flow of a db-gui-appis illustrated in Figure 2.1. Examples of db-gui-apps are described below.

- An enterprise application with a web front end: it receives user inputs from a web browser and references or updates data (e.g., customer information) in DB tables on the basis of its business logic. It finally displays the results on the web browser screen.
- A game application: it receives user inputs from a game screen and references and updates data (e.g., player and enemy information) in its memory and DB tables on the basis of its game logic. It finally displays the updated game screen.

The processes of receiving user inputs from the GUI, accessing the DB on the basis of application logic, and displaying the calculated results are repeatedly executed in a db-gui-app. Thus, these processes occupy most of the program execution time. Moreover, the behaviors of these processes greatly depend on the combination of various input patterns from the user, internal states of the program, and the DB state. Accordingly, many test patterns are required, and much time is taken for the tests. We thus focused

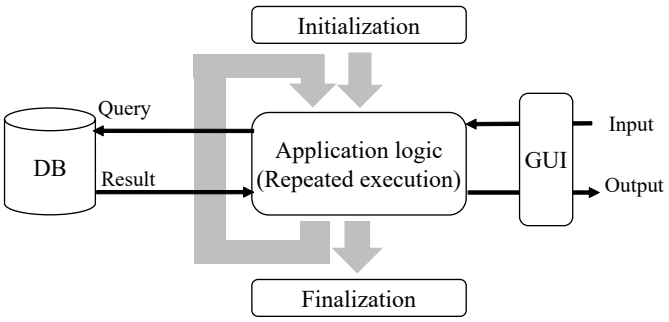


Figure 2.1: Typical Execution Flow of Database Applications with Graphical User Interface

on integration testing of such processes in which three modules (GUI, application logic, and DB) are integrated.

More specifically, integration testing was divided into two levels. The first level is for tests in which only one screen transition occurs per test case. This level mainly includes test cases from the viewpoints of the variations in input combinations and the variations in application logic behaviors. The second level is for tests in which multiple screen transitions occur. This level mainly includes test cases from the viewpoint of the variations in scenarios in accordance with the use cases of the software. Our study focused on the first level.

2.2 Current Status and Testing Challenges

There are two requirements for developing a practical and widely usable method for automating tests.

Requirement 1 It remains applicable when the software functions have evolved.

Requirement 2 It can be used independently of the software implementation technology.

Taking these requirements into consideration, we next outline existing studies and describe the four challenges to be tackled for each task in testing.

For unit testing, many studies have dealt with test design automation, mainly for improving code coverage using static and/or dynamic analysis of source code [63]. In particular, dynamic symbolic execution (DSE) [55] evolved remarkably over the ten years after it was invented in 2005 and is now in practical use as Microsoft Visual Studio IntelliTest [76]. Unit testing frameworks (e.g., JUnit in Java and NUnit in C#) are widely

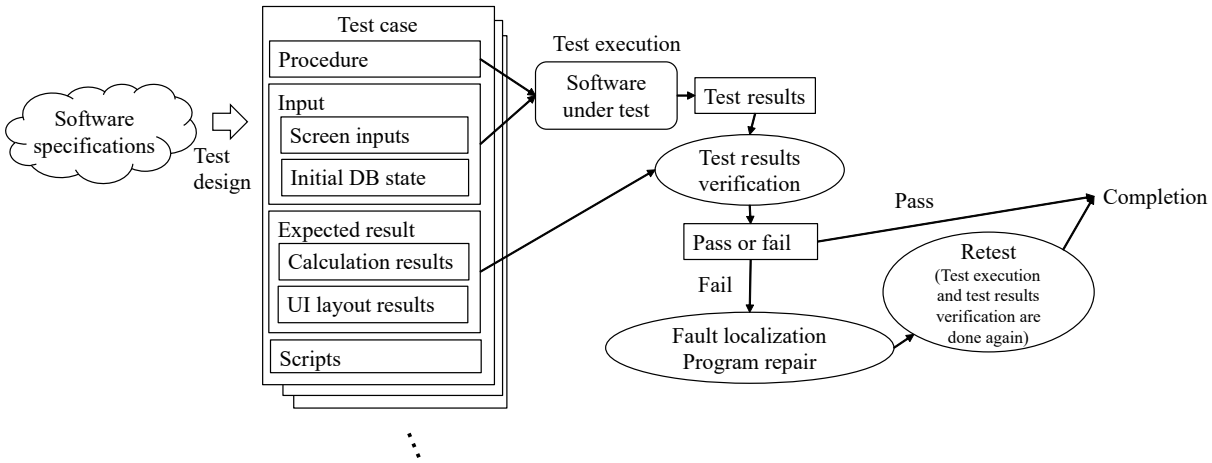


Figure 2.2: Integration Testing of Database Applications with Graphical User Interfaces

used at development sites for automating test execution and test results verification. In addition, since modules tested at the unit level are generally small, it is relatively easy to identify the cause(s) of detected bugs and remove them. On the other hand, integration testing is much more difficult to automate than unit testing; thus, various challenges remain in integration testing. Figure 2.2 shows how five of the tasks in Fig. 1.1 are performed in integration testing of a db-gui-app. We discuss each task focusing on integration testing.

2.2.1 Test Design

In integration testing, tests are conducted from the viewpoint of whether the software is implemented in accordance with its specifications. Therefore, test cases are created on the basis of the specifications. For each test case, it is necessary to prepare a procedure, input, expected results, and a script [63]. For a db-gui-app, they are as follows.

A procedure is a sequence of operations on the screen.

Input consists of screen inputs, and an initial DB state that is the appropriate precondition for the test case.

Expected results are the correct calculation results output by the application logic and the correct display results of the GUI layout created in accordance with the specifications.

A script is created for each test case if they are to be executed automatically.

Test cases should be created comprehensively from various viewpoints such as variations in the input combinations and variations in the behaviors of the application logic.

MBT [47] is a promising approach to automating test design. It can automatically generate comprehensive test cases on the basis of a model such as formal specifications, which are artifacts of the software design process. MBT satisfies both requirements above because it has the following functionalities.

- MBT can handle the evolution of software functions well. For example, when a model representing a specification is being modified, tests based on a new specification can be generated automatically from a new model.
- MBT can be used to develop a method to automate test design that is independent of the software implementation technology. This is done by using the concept of the platform-independent model (PIM) in model-driven engineering [27] and generating test cases from PIM.

Several existing methods [81] [80] [66] generate test cases on the basis of MBT; each test case has a procedure and screen inputs from a design model expressing the application logic and input definitions of the software. These methods generate screen input variations on the basis of boundary value analysis and equivalence partitioning. The screen input value types, including integer and string, are relatively simple. Since an initial DB state is a set of values of such types and must satisfy its DB schema constraints, it is more difficult to generate an appropriate state for each test case automatically. Several researchers [28] [24] have been able to generate the initial DB states, but there remain two challenges in testing industrial-level systems.

Challenge 1 Existing methods cannot generate initial DB states suitable for complicated business logic such as reading the DB more than once, searching the DB by partial string matching, or setting primary and foreign key constraints on the DB schema. We deal with this challenge in Chapter 3.

Challenge 2 Existing methods generate initial DB states for each test case one-by-one. However, there are two problems with this approach. The first is that switching initial DB states for each test case is too time-consuming when the generated initial DB states are used for testing. The second problem is that the total number of DB records tends to be large because many initial DB states are generated. As a result, the total size of the test data becomes large, which increases the cost of managing the data and the time needed to switch initial DB states for each test case. We deal with this challenge in Chapter 4.

The automatic generation of the expected result is called the "test oracle problem" [9] and is one of the most difficult problems in the field of test automation. Although it is possible to partially generate the expected result using information extracted from a model by using MBT, it is inherently difficult to generate it completely automatically. The reason is that, ultimately speaking, fully automatic generation of the expected result is equivalent to fully automatic implementation of the software. Therefore, an approach considered promising is to use the test result of the initial testing as the expected result for regression testing [9]; that is, the test results for the old and new versions of the same software are compared. Since this approach is simple and useful, it is used at many development sites. A challenge when verifying the test results in regression testing with this approach is described in Section 2.2.3. Test scripts for each test case can be generated from a design model, which can be converted from design documents used at development sites [68] [67] [71]. Moreover, methods for maintaining test scripts appropriately in response to software evolution have been proposed [37] [34].

2.2.2 Test Execution

The test cases created in test design are executed one-by-one for the software under test, and screen shot images are recorded as evidence. Test execution is the most automated area in integration testing. Open source software (OSS) tools for automatic test execution include Selenium[1], Appium[2], and Sikuli[3] [15], and there are many other commercial tools as well. Using these tools, the tester can automate test execution on various platforms such as Android, iOS, and Windows. Once scripts are prepared as input for such a tool, each test case can be executed automatically.

2.2.3 Test Results Verification

For each test case, the test result is compared with the expected result, and pass or fail is determined. In regression testing of a db-gui-app, it is necessary to ensure that the application screens are displayed correctly. This involves two confirmations: confirming whether the application logic works correctly and the calculation result is correct and confirming whether the screen elements are laid out correctly on every application screen. The former can be automated by using the test automation tool explained in Section 2.2.2 with suitable assertions in the scripts executed by the tool. In contrast, the latter

[1] `https://docs.seleniumhq.org`
[2] `http://appium.io/`
[3] `https://launchpad.net/sikuli`

requires that the tester carefully examine displayed layouts, so it is a difficult and time-consuming task. VRT[4] is a method for semi-automating the latter confirmation process. VRT detects differences between two screens of an application, typically corresponding ones before and after changes, on the basis of image information, structural information, and so forth for the screens. We call this approach to VRT in which two screenshot images are compared and only this information is used for confirmation *image-based VRT*. Since image-based VRT is applicable as long as screenshot images of application screens are available, it can be used independently of the operating environment (such as the OS or web browser) in which the application is executed. In addition, it can be easily used because many test automation tools provide a way to take screenshots of applications under test. Thus, there are many image-based VRT tools such as jsdiff[5] and BlinkDiff[6]. These tools compare two images in pixel units and highlight the pixels with differences to help the tester easily and clearly identify places with differences. However, there is a challenge regarding requirement 1.

Challenge 3 Existing image-based VRT systems simply compare two images in pixel units and highlight pixels with differences, so if there are changes that affect the entire screen (e.g., parallel movements of screen elements), a large number of unessential differences are detected, and the essential differences are buried within them. Therefore, it is difficult to handle software evolution. We deal with this challenge in Chapter 5.

2.2.4 Fault Localization

The programmer debugs a program by first identifying the causes of each bug detected in testing.

Classical breakpoint-based debugging, in which execution of the target program is suspended and the programmer observes the internal state of the program, is a practical and useful method and is provided as a standard debug interface for various programming languages. Therefore, it is used by many programmers [13]. This method is effective for processes in programs that are performed without user interaction, such as initialization and finalization (see Fig. 2.1. In contrast, the behavior of the application logic greatly depends on the combination of the various inputs from the user, the internal state of the program, and the DB state. Since the program executing the application logic has

[4] `https://github.com/mojoaxel/awesome-regression-testing`
[5] `https://github.com/kpdecker/jsdiff`
[6] `https://github.com/yahoo/blink-diff`

bidirectional interactive tasks and/or realtimeness, we refer to it as a interactive and/or realtime program.

Challenge 4 Breakpoint-based debugging is not suitable for debugging interactive and/or realtime programs for two reasons. First, since the timings and order of input events such as user operations are quite important, such programs do not behave as expected if execution is suspended at a breakpoint. Second, suspending a program to observe its internal states significantly degrades the efficiency of debugging. We deal with this challenge in Chapter 6.

2.2.5 Program Repair

The programmer needs to repair the program carefully and appropriately without introducing any software regressions, i.e., bugs, into the program by mistake.

Once the cause of a bug has been identified, it is usually relatively easy to remove the cause so that the bug is not reproduced. The most difficult part of fixing a bug is that it must be fixed with care to avoid introducing any new regressions. If the programmer modifies the program to fix a bug detected in integration testing, even a minor modification can negatively affect the whole program. This problem can be prevented by performing the regression tests properly [82]. If Challenges 1, 2, and 3 in test design and test results verification are met, regression testing after program modification can be performed efficiently. This enables the programmer to repair the program safely.

Chapter 3

Design-Model-based Initial DB State Generation

In this chapter, we describe a method to generate initial DB states from design model.

3.1 Introduction

When testing a db-gui-app, in addition to creating program inputs, it is necessary to create an initial DB state that is appropriate for each test case.

In practice, there are three approaches to realizing initial DB states.

1). Manual construction of the states from scratch

2). Usage of random number generation tools (e.g., using tools such as DBMonster[1], Visual Studio Database Edition[2], and PPDGen[3])

3). To use a subset of an actual DB used in the old system.

Unfortunately, manual construction is too time-consuming. Even if the states are automatically generated by approaches approach(2) and approach(3), we must remake or adjust the initial DB if it is fails to meet the preconditions of each test case.

For example, consider a system that identifies which employees should gain in-house training using the operations shown in Fig.3.1. The employee search system is designed to search employees who have worked for less than three years, and to support compound search keys such as an age or a section; it outputs 100 search results to each page. When

[1] http://dbmonster.kernelpanic.pl/

[2] http://www.microsoft.com/japan/msdn/vstudio/

[3] http://www.start-ppd.jp/

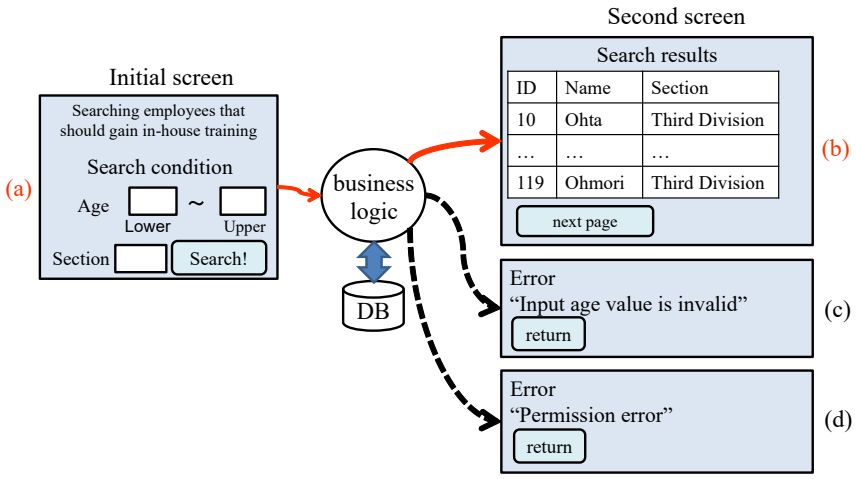


Figure 3.1: Example of Database Application

we test to check the transition from screen (a) to screen (b) in Figure 3.1, we must make an appropriate initial DB that has more than 101 records of employees that satisfy all of the following search keys; (i) the number of years working at the company is less than three years. (Fixed condition.) (ii) Boolean indicates whether the employee is still working in the company is true. (Fixed condition.) (iii) age is between `LowerAge` and `UpperAge`. (Variable conditions. `LowerAge` and `UpperAge` are given by the user.) (iv) section name to which the employee belongs is `SectionName`. (Variable condition. `SectionName` is given by the user.) If the Employee table of the initial DB generated by the approach(2) or the approach(3) does not contain more than 101 records, each of which satisfies the search keys (i),(ii),(iii) and (iv), the test case is invalid. The solution is to remake the state or add record(s) which meets the above search keys.

We categorize test cases as follows. Our study focuses on (1).

(1) Test cases in which the initial DB is read only,

(2) Test cases in which the initial DB is read and written to, and

(3) Test cases not requiring any initial DB.

Some studies have attempted to automatically generate appropriate initial DB states for (1) because referring access is frequently used in several systems [17], but none are truly practical as detailed in the next section. For (2), because the state of the DB changes every moment due to the update, it is necessary to consider the life cycle of the DB state when generating the initial DB state. This study does not cover (2) and we

discuss this in Section 7 as a future work. Emmi's method [25] dealt with (2), however, since this method aims to improve the coverage of the source code and adjusts the DB state and input values while executing a DB application, this is different from the purpose of our study aims to generate appropriate initial DB state based on the information of the specification. (3) does not require a variation of the initial DB state. In other words, it is a case where the test can be performed by preparing an arbitrary initial DB state (it may be empty). This study does not cover (3).

As mentioned above, our research scope is (1). Therefore, the definition of the appropriate initial DB state as the precondition of a test case in this research is "one that satisfies the DB schema and contains a certain number of records that meet the search conditions executed by the test case."

The contributions of the proposed method in this chapter can be summarized as follows.

- In test design, we propose a method called DDBGen (design-model-based initial DB state generator) that can create design models of industry-level enterprise systems from which appropriate initial DB states and input values can be created to generate a wide variety of test cases. Our study identified the constraints most frequently used in industrial-level enterprise systems; they include "multiple DB read," "PK,FK constraint," and "partial string matching."

- We propose an algorithm to generate an appropriate initial DB states by solving the constraints that should be satisfied in three steps. Step 1 solves the constraints on DB table size. Step 2 solves the constraints on string variable length and the constraints on integer variables. Finally, the constraints on character variables are solved.

- Since our design model can handle these constraints found, high initial DB generation rates were achieved in evaluations on three industrial-level enterprise systems. In addition, we propose another method to generate initial DB state shared by multiple test cases to reduce the number of times the initial DB state must be switched and reduce the total size of test data.

3.2 Related Work

There are two main types of existing methods to generate initial DB states suitable for each test case based on the information of the specification. We explain each method and its problems.

AGENDA [16] allows the tester to describe the preconditions that the initial DB state must satisfy for each test case and checks whether the initial DB state satisfies these preconditions before executing each test case. However, in this method, as the number of test cases increases, the preconditions are often not satisfied. In that case, the test need to adjust the initial DB state manually. If a precondition is not met, it is also conceivable to regenerate the initial DB state using random number generation tools and repeat the check until the precondition is satisfied. However, it is unlikely that a proper initial DB state can be generated in a realistic time by such a naive method.

In Willmor's approach [24], the user specifies the pre- and post-conditions that the initial DB state should satisfy for each test case; it then automatically adjusts an initial DB to yield the appropriate initial DB by adding or deleting records. Fujiwara's approach [28] automatically generates an initial DB that satisfies a precondition and a post-condition. These approaches have the following limitations when testing industrial-level enterprise systems.

- They cannot generate the initial DBs needed to confirm complicated business logic states such as reading the DB more than once. For example, if the DB must be accessed to check for read permission, then accessed again to search for employees, see Fig.3.1.

- They cannot handle the constraints often used in testing enterprise systems. For example, a column constraint that has both primary key attribute and foreign key attribute, and partial string matching used in search condition.

As a result, existing approaches are not strong enough to handle the wide range of conditions needed to generate appropriate initial DB states for each test case. In this case, we cannot get appropriate initial DB states.

3.3 Proposed Method

Our proposal is a new approach called DDBGen that generates test cases from a *design model* of the software specification, and the appropriate initial DB state for each test case. Figure 3.2 shows an overview of our approach. It is based on MBT [47], and so has the following features.

- On the basis of an existing design model [80] composed of *Process flow* and *Input definition*, the new design model adds the *DB schema* to process flow and input definition. Process flow can represent complex business logic states, such as reading

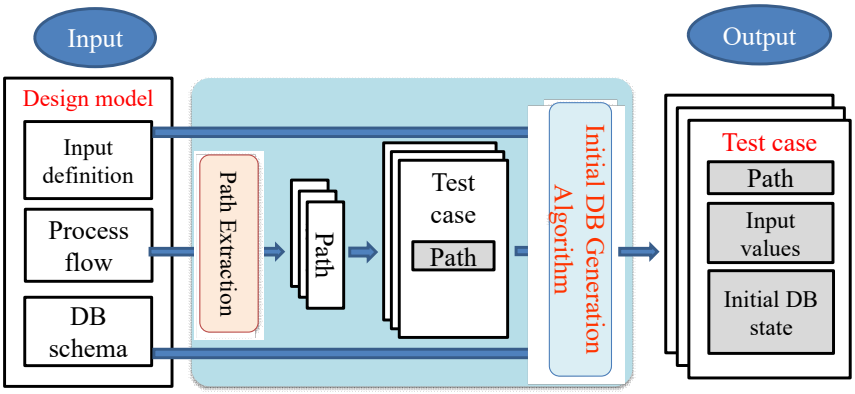


Figure 3.2: Overview of Proposed Method

the DB more than once. In addition, users can describe various conditions such as partial string matching and four arithmetic operations in each DB search condition to satisfy the majority of business logic states required by existing applications. Input definition can represent the inputs of screens and the domains of each input. The inputs can be used as DB search constraints to satisfy the business logic. The DB schema allows users to set primary key constraints and foreign key constraints.

- A test case generated by our approach corresponds to a *path* extracted from a process flow, and is composed of an initial DB state and input values. In our approach, we extract the conditions that have to be satisfied to generate the appropriate initial DB state and input values; the conditions are cast as a constraint satisfaction problem. We reduce the problem into several smaller problems that are tackled step-by-step by a constraint solver.

To exhaustively test the behaviors of the application implementing the business logic, we first extract all paths based on Decision/Condition Coverage [32] by using the existing method of [69] and generate test cases for each path. Next we generate initial DB states and input values for each test case. The following subsections detail the design model, the test cases and the algorithm for generating initial DB states.

3.3.1 Design Model

Figures 3.3, 3.4, and 3.5 elucidate the design model of the employee search system shown in Figure 3.1. In the business logic of the employee search system, first the consistency between `LowerAge` and `UpperAge` is checked (Figure 3.3(1)). Next, the permission of the

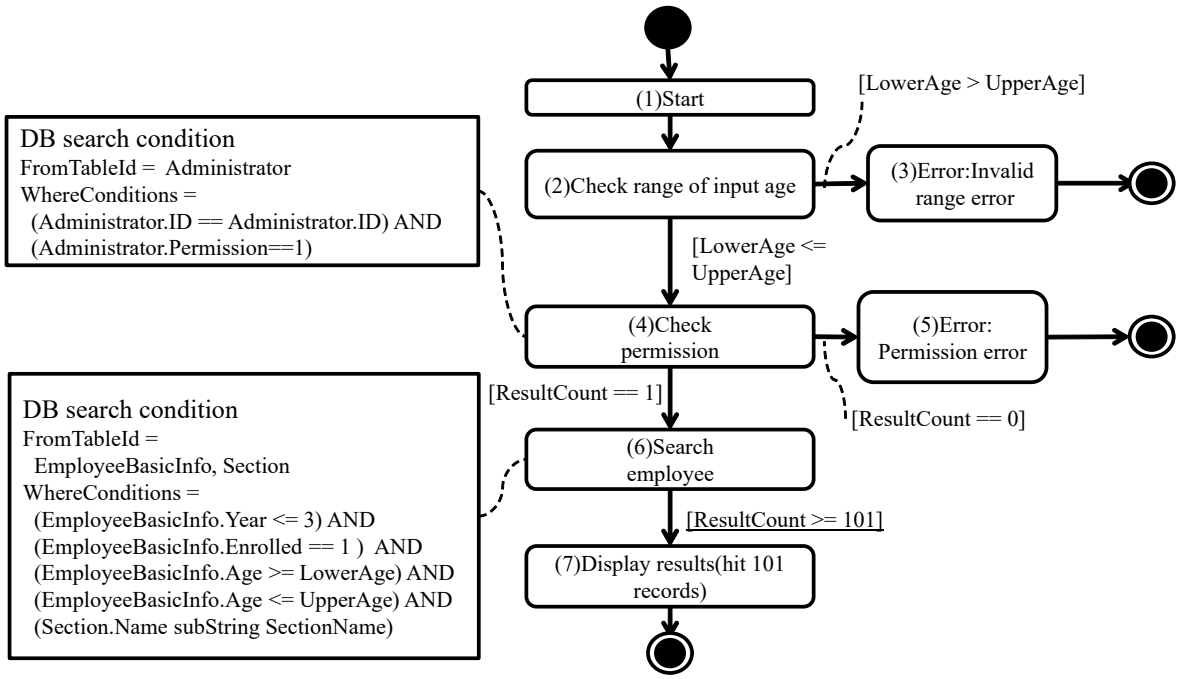
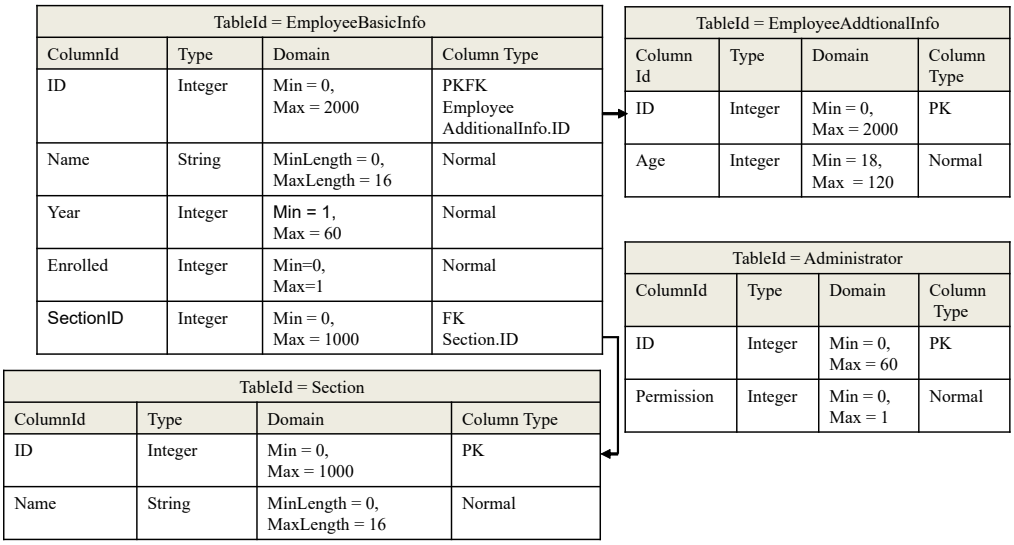


Figure 3.3: Design Model: Process Flow of Employee Search System

user who enters a search is confirmed (Figure 3.3(4)). Finally, the system identifies the employees that satisfy the search requirement (Figure 3.3(6)) if the user has permission to initiate the search. As seen above, users can describe multiple DB search conditions and can use various conditions such as partial string matching. The DB schema defines the DB tables such as `EmployeeBasicInfo` table, which contains employee information, and `EmployeeAdditionalInfo` table, which contains additional employee information. Users can use primary key constraints and foreign key constraints when defining these DB tables.

VariableId	Type	Domain
LowerAge	Integer	MinValue = 18, MaxValue = 120
UpperAge	Integer	MinValue = 18, MaxValue = 120
SectionName	String	MinLength = 0, MaxLength = 16
AdminID	Integer	MinValue = 50, MaxValue = 60

Figure 3.4: Design Model: Input Definition of Employee Search System



TableId = EmployeeBasicInfo

ColumnId	Type	Domain	Column Type
ID	Integer	Min = 0, Max = 2000	PKFK Employee AdditionalInfo.ID
Name	String	MinLength = 0, MaxLength = 16	Normal
Year	Integer	Min = 1, Max = 60	Normal
Enrolled	Integer	Min=0, Max=1	Normal
SectionID	Integer	Min = 0, Max = 1000	FK Section.ID

TableId = EmployeeAddtionalInfo

Column Id	Type	Domain	Column Type
ID	Integer	Min = 0, Max = 2000	PK
Age	Integer	Min = 18, Max = 120	Normal

TableId = Administrator

ColumnId	Type	Domain	Column Type
ID	Integer	Min = 0, Max = 60	PK
Permission	Integer	Min = 0, Max = 1	Normal

TableId = Section

ColumnId	Type	Domain	Column Type
ID	Integer	Min = 0, Max = 1000	PK
Name	String	MinLength = 0, MaxLength = 16	Normal

Figure 3.5: Design Model: DB Schema of Employee Search System

Table 3.1: Definition of Design Model

Id	Definition		
		Design Model	
0	⟨ Design Model ⟩	::=	⟨Process Flow⟩+ ⟨Input Definition⟩+ ⟨DB Schema⟩
		DB Schema	
1	⟨ DB Schema ⟩	::=	⟨ Table ⟩ *
2	⟨Table ⟩	::=	TableId:String ⟨Column⟩+
3	⟨Column⟩	::=	ColumnId:String ⟨TypeAndDomain ⟩ ⟨KindOfVolumn⟩
4	⟨TypeAndDomain⟩	::=	IntegerType ⟨IntegerDomain⟩
5	⟨IntegerDomain⟩	::=	MinValue:Integer MaxValue:Integer
6	⟨StringDomain⟩	::=	MinLength:Integer MaxLength:Integer
7	⟨KindOfColumn⟩	::=	Normal | PK | (FK ⟨ColumnReference⟩) | (PKFK ⟨ColumnReference⟩))
8	⟨ColumnReference⟩	::=	TableId:String ColumnId:String
		Process Flow	
9	⟨Process Flow⟩	::=	⟨Node⟩+ ⟨Edge⟩*
10	⟨Node⟩	::=	NodeId:String Text:String NextEdgeId:String*
11	⟨Edge⟩	::=	EdgeId:String NextNodeId:String (⟨GuardCondition⟩? | ⟨DBSearchCondition⟩?)
12	⟨DBSerachCondition⟩	::=	FromTableId:String+ ⟨WhereClause⟩ ⟨ResultCondition⟩
13	⟨WhereClause⟩	::=	⟨Conditions⟩
14	⟨ResultCondition⟩	::=	ResultRecordCount:Integer
15	⟨GuardCondition⟩	::=	⟨Conditoons⟩
16	⟨Conditions⟩	::=	⟨Constraint⟩+
		Input Definition	
17	⟨Input Definition⟩	::=	⟨Input⟩+
18	⟨Input⟩	::=	⟨InputId:String⟩ ⟨TypeAndDomain⟩

Table 3.2: Definition of Constraints on Where Clauses and Guard Conditions

Id	Definition		
1	⟨Constraint⟩	::=	⟨IntegerConstraint⟩ | ⟨StringConstraint⟩
2	⟨IntegerConstraint⟩	::=	⟨Expression⟩ ⟨ComparedOperator⟩ ⟨Expression⟩
3	⟨Expression⟩	::=	⟨Term⟩ | ⟨Expression⟩ "+" ⟨Term⟩ | ⟨Expression⟩ "-" ⟨Term⟩
4	⟨Term⟩	::=	⟨Factor⟩ | ⟨Term⟩ * ⟨Factor⟩ | ⟨Term⟩ / ⟨Factor⟩
5	⟨ComparedOperator⟩	::=	"⟩" | ">=" | "⟨" | "<=" | "==" | "! ="
6	⟨Factor⟩	::=	ConstantValue:Integer | ⟨InputReference⟩ | ⟨StringLength⟩ | ⟨ColumnReference⟩ | "(" ⟨Expression⟩ ")"
7	⟨StringLength⟩	::=	Length (⟨StringFactor⟩)
8	⟨StringConstraint⟩	::=	⟨StringFactor⟩ ⟨StringCompareOperator⟩ ⟨StringFactor⟩
9	⟨StringCompareOperator⟩	::=	SubString | EqualsString | NotSubString | NotEqualsString
10	⟨StringFactor⟩	::=	ConstantValue:String | ⟨OnputReference⟩ | ⟨ColumuReference⟩
11	⟨InputReference⟩	::=	VariableId:String
12	⟨ColumnReference⟩	::=	TableId:String ColumnId:String

Table 3.1 shows the definition of the design model, and Table 3.2 shows the definitions of the Where clause conditions included in the DB search conditions and the constraints that can be described in the guard conditions.

3.3.2 Test Case

Each test case generated by our approach consists of path, initial DB state, and user-generated input values.

The initial DB state satisfies the constraints of the DB schema and contains enough records that satisfy the search conditions in each read access, where the input values can be used as parameters of search conditions. The input values satisfy both the guard conditions in the path and the domains specified in the input definition. As a result, it is guaranteed that the initial DBs and the input values generated by our approach are appropriate for each test case.

Figure 3.6 shows an example of a test case for the employee search system. The test case corresponds to the path composed of nodes (1), (2), (4), (6) and (7) in Figure 3.3. The test case is used to check the transition from screen(a) to screen(b) in Figure 3.1.

In Figure 3.6, the initial DB state contains records in not only `EmployeeBasicInfo` table `Administrator` table and `Section` Table, but also `EmployeeAdditinalInfo` table. The first three tables are directly referenced by the DB search conditions in the path of the test case. The `EmployeeAdditinalInfo` table is not explicitly described in the FROM clauses of the DB search conditions but is needed since its records are essential for ensuring referential integrity. Referential integrity means that the foreign key in any referencing table must always refer to a valid row in the referenced table.

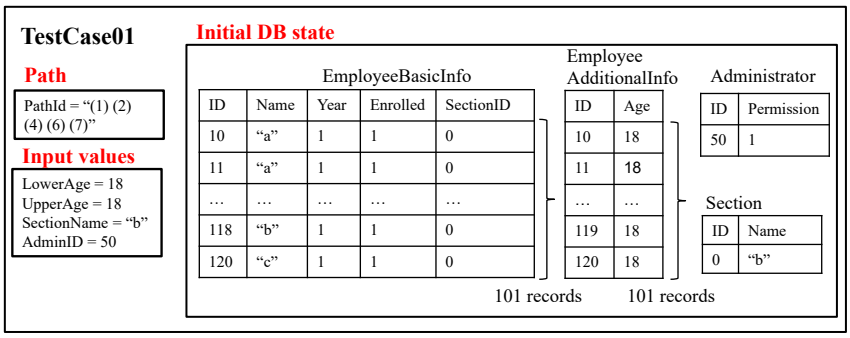


Figure 3.6: Test Case of Employee Search System

Table 3.3: Test Cases and Variables

Id	Definition		
1	⟨TestCase⟩	::=	PathId:String ⟨InitialDatabaseStateVariable⟩ ⟨InputVariable⟩+
2	⟨InitialDatabaseStateVariable⟩	::=	⟨TableVariable⟩+
3	⟨TableVariable⟩	::=	TableId:String ⟨ RecordCoountVariable ⟩ ⟨RecordVariable⟩*
4	⟨RecordCoountVariable⟩	::=	⟨IntegerVariable⟩
5	⟨RecordVariable⟩	::=	Index:Integer ⟨FieldVariable⟩+
6	⟨FieldVariable⟩	::=	ColumnId::String (⟨IntegerVariable⟩ | ⟨StringVariable⟩)
7	⟨InputValue⟩	::=	InputId:String (⟨IntegerVariable⟩ | ⟨StringVariable⟩)
8	⟨StringVariable⟩	::=	⟨StringLengthVatiable⟩ ⟨CharVariable⟩*
9	⟨StringLengthVatiable⟩	::=	⟨IntegerVariable⟩
10	⟨IntegerVariable⟩	::=	Value:Integer
11	⟨StringVariable⟩	::=	Index:Integer Value:Char

3.3.3 Algorithm of Generating Initial Database State and Input Values

This section describes the algorithm that generates initial DB states and input values. Some studies [54] [80] developed approaches to extract test cases and generate input values for each test case. First, they extract paths from a process flow, and then create test cases on the basis of each path. Finally, they consider the input values as variables and solve for the variables. In our approach, the variables to be solved are both the initial DB state (each table, each record and each field) and the input values. We will call the constraints extracted from the design model *the constraint set* and call the variables *the variable set*. Table 3.3 shows the definition of the variable set. We consider the constraint set and the variable set as one simultaneous constraint, which is solved by applying the following policies.

- We use the Arithmetic Theory of Satisfiability Modulo Theories (called SMT here-

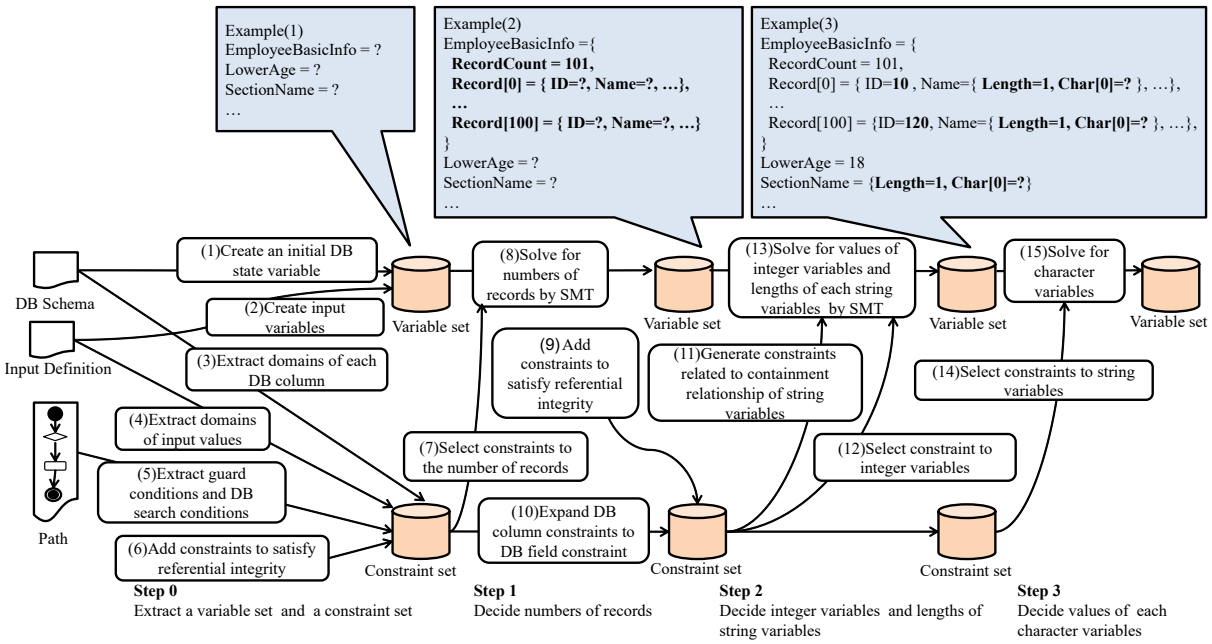


Figure 3.7: Overview of Algorithm that Generates Initial Database State and Input Values

after) to resolve the constraints into integer variables; the constraints are treated as a simultaneous inequality.

- SMT can handle integer-type simultaneous inequalities and Boolean expressions, but cannot directly handle data structures and constraints [23]. Considering a DB table as an array of records, our solution is to solve the constraints of DB tables sizes to decide these sizes, then solve the constraints of each record. We can handle string variables in the same way as DB tables by considering a string as an array of characters.

The above policies yield the initial DB and input values by solving the constraints in the constraint set in a step-by-step manner. Figure 3.7 overviews this process. Step 1 solves the constraints on DB table size. Step 2 solves the constraints on string variable length and the constraints on integer variables. Finally, the constraints on character variables are solved.

We detail steps 0, 1, 2, and 3 in Figure 3.7 in the following subsections.

Step 0: Extracting Variable Set and Constraint Set from Design Model

Step 0 is composed of the following procedures (Figure 3.7 (1) to (6)).

(1) Creates an initial DB state variable based on the DB schema and add it to the variable set.

(2) Creates input value variables based on the input definition and add them to the variable set.

(3) Extracts domains of each DB column from the DB schema and add them to the constraint set.

(4) Extracts domains of each input value from the input definition and add them to constraint set.

(5) Extracts guard conditions and DB search conditions from the path and add them to constraint set.

(6) To satisfy referential integrity, adds constraints related to the numbers of records to the constraint set.

To satisfy referential integrity, a DB table referred to by a foreign key of another DB table must contain at least one record. Hence, first we collect the DB table variables (t_{target}) that are directly or indirectly referred to by the foreign keys of other DB table variables used in the DB search conditions in the path; next we add constraint "$Count(t_{target}) >= 1$" to the constraint set. If a DB column of DB table variable ($t_{referring}$) has both a primary key constraint and a foreign key constraint, the DB table variable ($t_{referred}$) referred to by the column of $t_{referring}$ must contain more records than the number of records of $t_{referring}$. Hence, we add constraints "$Count(t_{referring}) <= Count(t_{referred})$" to the constraint set.

Step 1: Decide Number of Records

Step 1 is composed of the following procedures (Figure 3.7 (7) to (10)).

(7) Select constraints ($\mathbb{C}_{records\_number}$) related to the number of records from the constraint set.

(8) Solve ($\mathbb{C}_{record\_number}$) by SMT to decide the number of records of each DB table variable, and remove $\mathbb{C}_{record\_number}$ from the constraint set.

(9) To satisfy referential integrity, add constraints related to DB fields to the constraint set.

(10) Expand constraints ($\mathbb{C}_{column}$) related to DB columns to other ones related to DB fields, and remove $\mathbb{C}_{column}$ from the constraint set.

In procedure (8), considering constraints on the numbers of DB records as a simultaneous inequality, we use SMT to solve for the number of DB records.

In procedure (9), to ensure that each primary key variable ($pk_1, ..., pk_n$) in a DB table variable is unique, we add constraints on integer type, "$pk_1 < pk_2$," ... , "$pk_{n-1} < pk_n$," to the constraint set if the primary keys are integer type and, we add constraints on string type, "pk_1 *subString* c_1," ...,"pk_n *subString* c_n," where $c_1, ..., c_n$ are differ from each other, to the constraint set if the primary keys are string type.

To ensure that DB field variable (fk) matches the DB field variable that is referred to by fk, we arbitrarily select one DB field variable (pk) referred to by fk, and generate new constraints "$fk == pk$" (in the case of integer type) or "fk *equalsString* pk" (in the case of string type). Note that if fk has a primary key constraint in addition to a foreign key constraint, we select different DB field variables pk for each fk.

In procedure (10), we convert the constraint "c *Operator* x" (c is a DB column, and x is a constant value or input value),which is extracted from the DB search conditions and the domains of the DB schema, to other constraints "f_1 *Operator* x,"...,"f_n *Operator* x" ($f_1, ...f_n$ are DB fields that correspond to c). These covered constraints are added to the constraint set.

DB table size is fixed at the conclusion of Step 1. Therefore, the state of the constraint set transitions from the state of Figure 3.7 Example(1) to the state of Figure 3.7 Example(2).

Step 2: Decide Integer Variables and Lengths of String Variables

Step 2 is composed of the following procedures (Figure 3.7 (11) to (13)).

(11) Converts constraints $\mathbb{C}_{string\_length}$ into lengths of string variables by considering the containment relationship of string variables, and add $\mathbb{C}_{string\_length}$ to the constraint set.

(12) Selects constraints $\mathbb{C}_{integer}$ related to integer variables from the constraint set. Note that $\mathbb{C}_{integer}$ includes $\mathbb{C}_{string\_length}$.

(13) Solves $\mathbb{C}_{integer}$ by SMT to decide the value of each integer variable, and remove $\mathbb{C}_{integer}$ from the constraint set.

To ensure that a summation of substring variable lengths ($s_1, ..., s_n$) does not exceed the length of the string variable (s), where s contains each s_i, we generate the constraint

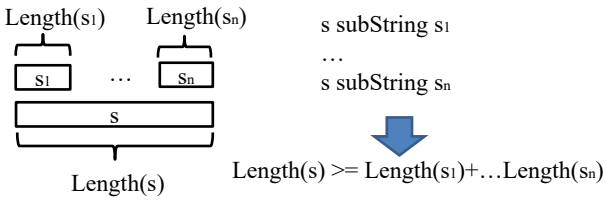


Figure 3.8: Generate Constraints from Containment Relationship of Strings

"$Length(s) \geq Length(s_1) + ... + Length(s_n)$" based on the containment relationship of the string variables (see Figure 3.8) and add the constraint to the constraint set. If string variable x must equal string variable y, we generate the constraint "$Length(x) = Length(y)$" and add it to the constraint set.

The values of the lengths of each string variable and the integer variables are decided after step 2. Therefore, the state of the constraint set transitions from the state of Figure 3.7 Example(2) to the state of Figure 3.7 Example(3).

Step 3: Decide Character Variables

Step 3 is composed of the following procedures (Figure 3.7 (14) to (15)).

(14) Selects constraints ($\mathbb{C}_{string}$) related to string variables.

(15) Decides values of each character variable based on $\mathbb{C}_{string}$, and remove $\mathbb{C}_{string}$ from the constraint set.

In procedure (15), if there is a correspondence relation among string variable $s0$ and other string variables $s_1, ..., s_n$, as shown in Figure 3.9, we consider the correspondence in determining the values of each character variable.

After step 3, the constraint set is empty and all values of the variable set are decided. The state of the constraint set transitions from the state of Figure 3.7 Example(3) to that of Example(4) as shown in Figure 3.6 , and we get the appropriate initial DB state and input values such as Figure 3.6.

3.4 Evaluation

3.4.1 Measure

The goal of our research is to generate appropriate initial DB states for a wider variety of test cases for db-gui-apps. Therefore, the performance measure is the variety of test cases for which we can generate appropriate initial DB states (and input values).

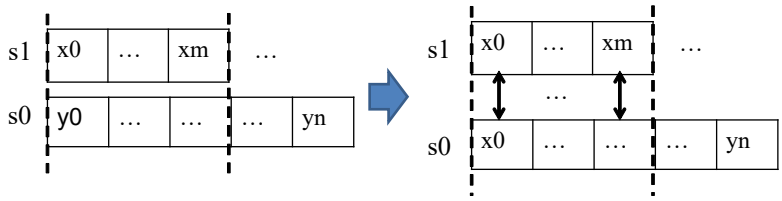


Figure 3.9: Decide Values of Character Variables

3.4.2 Procedures

We implemented a prototype tool based on our proposed approach. We used SMT Solver Z3[4] when resolving constraints into integer variables in Step 1 and 2 of our algorithm.

Since our approach is focused on the test cases that require only DB reads, we selected some of the functions of three industrial-level enterprise systems (A, B and C, see Table 3.4). The three systems have different levels of complexity in terms of DB search conditions and DB schema as follows. System A is the web front-end of a network equipment management system for operators. It uses relatively simple DB search conditions, such as search by IP address. System B is a scheduler system that manages schedules for each user and provides a date range search function. System C is a data mining system where users can set various data search conditions. Systems B and C offer more complex DB search conditions and employ more complex DB schemas than system A.

The prototype tool accepts XML design model files as input. Hence first we extracted paths from an existing design document of each system by using an existing method [69]. Second, we manually made XML design models based on the paths, and then added DB search conditions and DB schema, which are extracted from the design documents, to the design models. Finally, we input the XML design models to the prototype tool to generate initial DBs and input values. All tasks described to above were performed by a developer experienced in enterprise system creation. The prototype tool was run on a Windows Vista Ultimate SP2 machine with Intel Core i7 (3.0GHz) and 6GB of main memory.

The prototype tool handles only integer, character string, and table data types, and cannot handle other data types. Therefore, the enumeration type was treated as the string type, and the other basic data types were treated as the integer types.

Table 3.4: Category of Test Case

	Category	Number of Test Cases		
		A 4 tables 11 screens	B 12 tables 12 screens	C 40 tables 29 screens
(1)	Requiring initial DBs (read access)	74	83	44
(2)	Requiring initial DBs (read and write access)	7	4	9
(3)	Not requiring specific initial DB	147	292	108
	Total	228	379	161

Table 3.5: Evaluation Results

	A	B	C
Number of test cases our approach focus on	74	83	44
Generation rate of initial DB state	100%(74/74)	72%(60/83)	89%(39/44)
Generation time	0.1s/a test case	0.1s/a test case	0.1s/a test case

3.4.3 Result

As shown in Table 3.4, we categorize the test cases extracted from each design model, into three groups as noted in Section 3.1. Our approach focuses on the test cases of Table 3.4(1) and not those of Table 3.4(2). The test case of Table 3.4(3) do not require specific initial DB states and includes test cases to check for errors such as DB connection errors, file access failure errors, session errors, and invalid input value error.

The results of the evaluation are shown in Table 3.5. In each of A, B and C, our

Table 3.6: Use Frequencies of Characteristics Constraints We Propose

	A	B	C
Total	74	60	39
Multiple read DB	39/74(47%)	39/60(65%)	10/39(26%)
PK FK constraint	28/74(38%)	0/60(0%)	0/39(0%)
Partial string matching	12/74(16%)	10/60(17%)	0/39(0%)

[4] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

method was applicable to 100% ,72% and 89% of the test cases targeted by our study respectively. Our method could generate appropriate initial DB state for them.

Each generation time of an initial DB state is about 0.1 second per a test case. Even if it is considered to apply to 58,000 test cases of a 1000KL-scale software (the number of test cases in integration testing is about 58 per 1KL [1]), it takes less than 2 hours to generate initial DB states. Therefore, this is within the time that can be conducted as a night batch processing in practice.

The usage frequencies of common constraints; these constraints, shown in Table 3.6, are significant with regard to determining initial DB states. Part of the originality of our research lies in the recognition of the importance of these constraints.

3.4.4 Discussion

Constraints Could Not Be Satisfied

The test cases for systems A, B, and C, which we used for the evaluation, were created to check transitions from one screen to another. They had similar DB access patterns. First, permission to access the DB was checked. Then, a main DB search was executed in which each of the DB accesses referred to a different DB table. Therefore, many test cases can be generated with our approach because it can handle multiple DB reads. In addition, as shown in Table 3.6, it is effective for generating more appropriate initial DB states to handle DB columns that have both primary key and foreign key constraints and partial string matching.

However, design information of systems B and C has a more complex DB schema than system A. Each schema includes composite key constraints to express one-to-many correspondences such as schedules for each user. As a result, our design model could not fully handle these schema, and the generation rates were not 100%. Satisfying composite key constraints is a challenge remaining for future work.

Backtracking

Our algorithm does not perform backtracking, so if backtracking is required in step 1, in which the number of DB records is determined, or in step 2, in which the string lengths are determined, an initial DB state cannot be generated for a test case. Therefore, the initial DB states in our evaluation were generated without backtracking.

In step 1, a certain value is determined for each number of DB records, and these values are used in step 2. Therefore, for example, if a table contains an integer-type primary key column in which the domain is between 1 and 10 and the number of records

in the table is determined to be 100 given the constraint that "the number of records must be 5 or more" in step 1, 100 unique values cannot be generated for the fields of each record corresponding to that column in step 2. In this case, the number of records must be redetermined by backtracking to find a solution other than 100. However, this did not occur in our evaluation because Z3 was designed to select the smallest absolute value from among the candidate solutions.

In the determination of the string lengths in Step 2, for example, string variables x, y, z, there is no solution for the constraint "$Length(x) + Length(y) \leq Length(z)$" created on the basis of two constraints, "z *subString* x, z *subString* y" and "$Length(x) = 1, Length(y) = 1, Length(z) = 1$," extracted from the design model. In this case, the two constraints on the string lengths of x, y, z must be backtracked and solved using another constraint, "$Max(Length(x), Length(y)) \leq Length(z)$." Fortunately, this situation is unlikely because the restriction on the string length of each input field on the screen is rarely so strict.

If backtracking is needed, a limit can be set on the time taken to generate the initial DB state of each test case, and as many initial DB states as possible can be generated within the limited time.

Generation Time

Unlike the random number generation methods, which generate a large number of DB records, our method generates the minimum number of DB records required for each test case. Therefore, the generation times were not excessively long. For all three systems, most test cases did not require an excessive number of records. For example, only 1 record was required to test the login processing or the checking access authority processing in several test cases, and only about 10 to 15 records were required to test the function searching for DB records that meets some conditions in other test cases.

Quality of Generated initial DB state

The initial DB states automatically generated by our method have two advantages compared with ones created manually.

- Generation of only the minimum required number of records for each test case facilitates identification of the cause of a bug related to the data in the DB. In contrast, with manual creation, the tester often creates an initial DB state that can be used as a precondition for multiple test cases. Then, the number of records in the DB tends to increase. Therefore, it may take an excessive amount of time to identify the records related to the bug.

- The creation of test data does not depend on the tester's skills. Z3 can uniquely determine the solution for the same constraint, so DDBGen can uniquely obtain the same initial DB states regardless of which tester uses it as long as the design model is the same.

They also have two shortcomings.

- Since the initial DB state generated by DDBGen contains only those records that match the search condition, it cannot be used to check whether records that do not match the search condition are erroneously included in the search results. When a tester creates test data manually, there is often at least one record that does not match the search condition. It would be helpful in generating an initial DB state to check for that condition by generating extra records that do not match the search condition. This could be done on the basis of logical negation of the constraint that satisfies the search condition.

- There are few variations in the values in each field in the initial generated DB state. Z3 chooses the smallest absolute value as the solution when there are multiple candidate solutions. As a result, many fields often have the same value, as shown in Figure 3.6. It is useful to have variations in the test data values such as the boundary values. A promising approach is to utilize boundary value constraints created from the design model by the existing methods [80] [66].

3.5 Chapter Summary

In this chapter, we proposed DDBGen that can create design models of industry-level enterprise systems from which appropriate initial DBs and input values can be created to generate a wide variety of test cases. Our study identified the constraints most frequently used in industrial-level enterprise systems; they include "multiple DB read," "PK,FK constraint" and "partial string matching." Since our design model can handle these constraints found, high initial DB generation rates were achieved in evaluations on three industrial-level enterprise systems.

Future work is to evaluate our approach with other enterprise systems and to improve our approach to handle more complex constraints, such as composite key constraints, to generate appropriate initial DB states and input values for an even wider variety of test cases.

Chapter 4

Reducing Number and Size of Initial DB State

In this chapter, we describe a method for reducing the number and the size of initial DB states based on the method introduced in the previous chapter.

4.1 Introduction

The methods introduced in the previous chapter and Fujiwara's method [28] generate initial DB states and input values for each test case one-by-one. Hereinafter, we call these methods *one-by-one methods*. The cost of constructing the test data manually for many test cases can be reduced by using the *one-by-one methods*. However, when the test cases and test data are generated by one-by-one methods, the following problems arise in the test execution phase.

1. The problem of switching initial DB states: The initial DB states are generated for each test case one-by-one, so the tester must switch the initial DB states for each test case in the test execution phase. In practice, there are thousands of test cases, making it impractical to switch the initial DB states for each test case. Test execution can be partially automated; JUnit makes it especially easy to automate unit testing. However, it is not easy to automate the entire process of test execution considering integration testing of enterprise systems, and many manual processes remain. Clearly it is important to reduce the number of times the initial DB states are switched to reduce the cost of test execution. In addition, even if test execution can be fully automated, there is another problem that test execution time is lengthened by switching them over and over. This problem becomes more serious as the size of individual initial DB states increases.

2. The problem of the size of initial DB states: The number of DB records tends to be large because initial DB states must be generated for each test case, and there are many test cases in practice. As a result, the total size of the test data becomes large. In recent software development, it is often necessary to manage versions of test materials upon frequent releases, and to frequently send and receive data to and from offshore companies commissioned to execute tests. Therefore, as the data size of the initial DB states increases, the data management cost also increases.

For example, consider creating the test cases suitable for the enterprise system shown in Figure 4.1; the goal is validate the employees management system. Figure 4.1 shows two test cases. Test case (1) (Figure 4.1(1)) validates the function to locate employees who should receive in-house training. This function is designed to search for employees who have worked for less than two years and who belong to the section whose id is given by user as input. It outputs five search results per page. Test case (2) (Figure 4.1(2)) validates the function to locate the employees who should receive a complete medical checkup. This function finds employees who are over 25 and belong to the section whose id is given by user as input. It outputs four search results per page. Both test cases need appropriate initial DB states and inputs if the testing is to be effective. For instance, test case (1) needs section id as an input and an initial DB state that has at least six employee records that satisfy the following search keys (i) worked for less than two years. (Fixed condition.) (ii)belongs to the section whose id is `SectionName`. (Variable condition. `SectionId` is given by the user.) Figure 4.2 shows an example of automatically generating the test case and test data of the employee management system shown in Figure 4.1 by using the approaches proposed in our previous work. The approach generates initial DB states for each test case one-by-one as shown in Figure 4.2, where each initial DB states satisfies the preconditions of the corresponding test case. For example, Figure 4.2 shows that existing approaches generate two initial DB states and a total of eleven records to cover the two test cases. The tester has to switch the initial DB states to run each test case. If the test cases can share the same initial DB states, the tester would not need to switch (the first problem). In addition, these eleven records are somewhat redundant because only six records are needed if these test cases share the same initial DB state (the second problem).

The purpose of our research is to solve the above two problems, and to generate initial DB states that are used and shared by multiple test cases as shown in Figure 4.3, not generate one for each test case one-by-one as shown in Figure 4.2. Our proposal can reduce the number of initial DB states and the total number of DB records compared to the existing approach, hence it can reduce the number of times the initial DB state

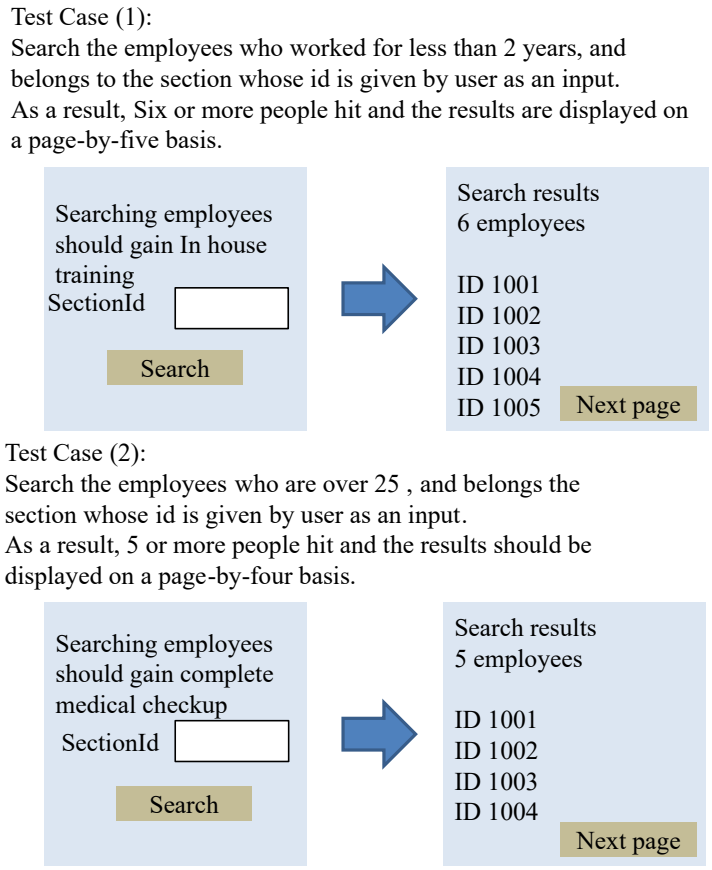


Figure 4.1: Example Test Cases

must be switched and reduce the total size of test data. As a result, our approach can reduce the cost of test execution. To measure the effect of our approach, we adopt two evaluation indexes. The first is *the number of initial DB states* and the second is *the total number of DB records*.

Our study focus on the test cases that perform DB referencing access rather than DB updating access for the same reasons as mentioned in the previous chapter.

The two main contributions of the proposed method in this chapter are as follows.

1. We propose a framework for generating initial DB states that are shared by multiple test cases, and list the three challenges (test case grouping, DB record arrangement, and initial DB state constraint generation) in the framework and also give simple and reasonable solutions to each challenge as the proposed method called DDB-GenMT (design-model-based initial DB state generator for multiple test cases).

2. Using industrial-level enterprise systems as case studies, we confirm that our ap-

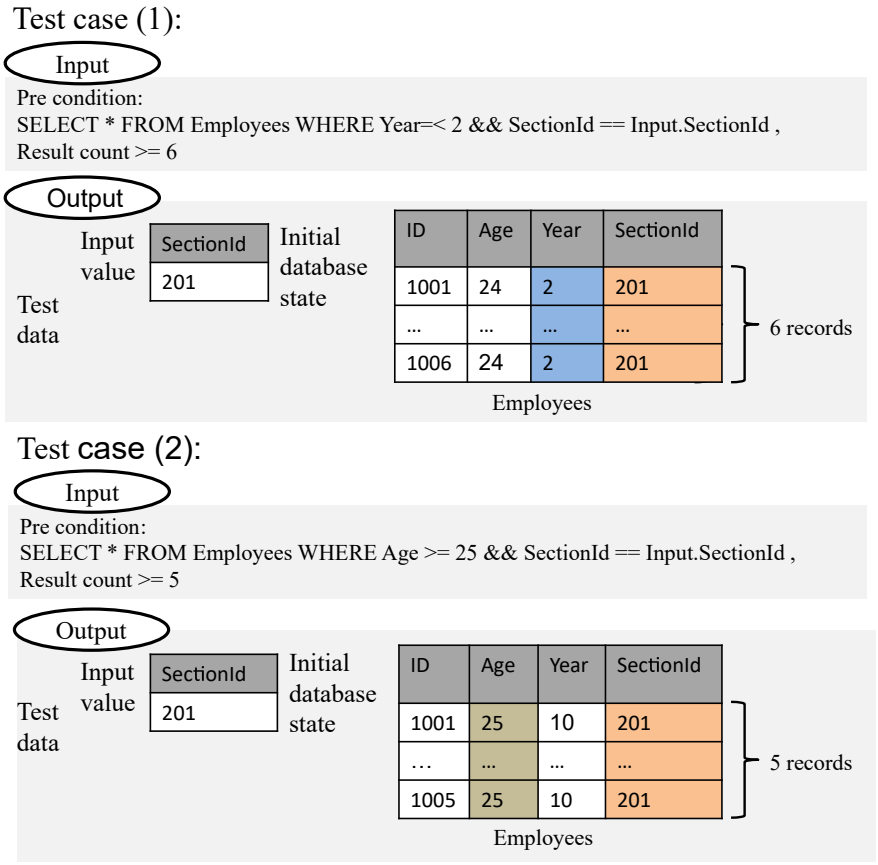


Figure 4.2: Initial Database States Generation by Existing Methods

proach reduces the number of initial DB states by 23%, and the total number of DB records by 64% compared to an existing one-by-one method.

4.2 Related Work

From the viewpoint of how to reduce the number of initial DB states and the number of DB records, we divided the existing methods for generating initial DB states into the two types.

The first type is one-by-one methods which includes the method introduced in previous chapter and Fujiwara's method [28]. Since these methods generate initial DB states for each test case, the number of initial DB states and the number of DB records tend to increase.

The first type covers other variants. These approaches adjust the initial DB state

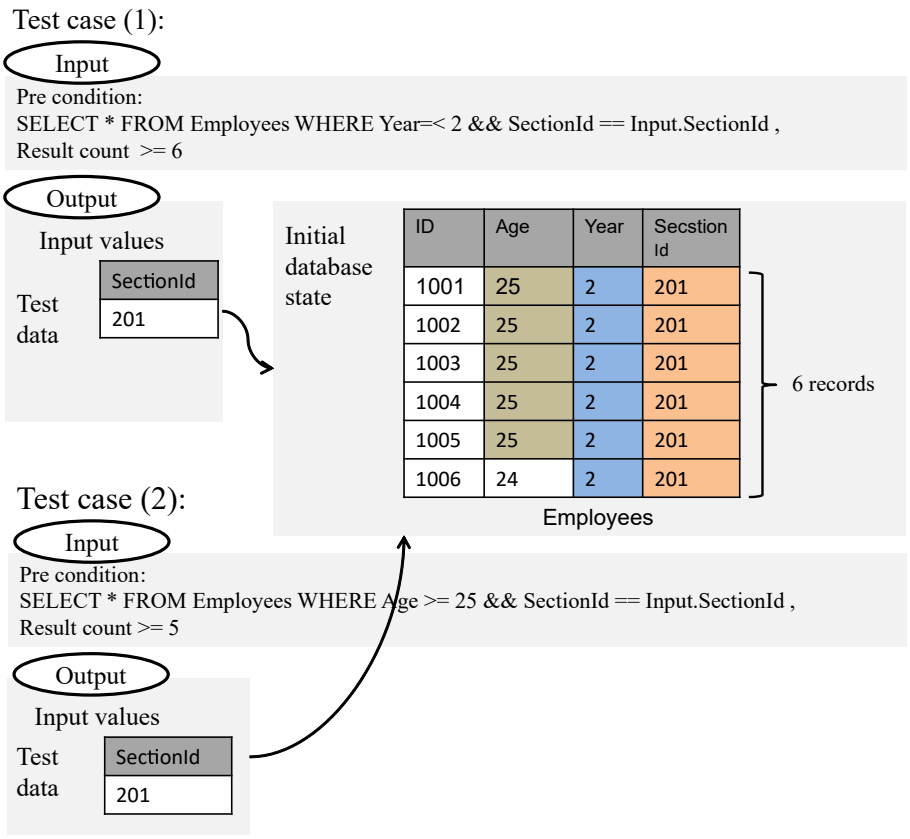


Figure 4.3: Initial Database States Generation by Proposed Method

during each test case execution. In Willmor's method [24], the user specifies the pre- and post-conditions that the initial DB state should satisfy for each test case; it then automatically adjusts the initial DB by adding or deleting records. Emmi's method [25] focuses on the methods that refer to DBs in the program and aims to achieve high path coverage. They repeatedly adjust the values of initial DB states and inputs to make these values cover a path that has yet to be executed, in the program by using the DSE. These methods also essentially provide initial DB states for each test case one-by-one.

The second type use an existing initial DB state such as one generated by random number generation tools, and adjust only inputs to provide an appropriate initial DB state and inputs for each test case. Pan's methods [48] [49] also aims to achieve high path coverage just like Emmi's approach. It repeatedly adjusts only the values of inputs to make these values activate a path that has yet to be executed by using the DSE. This methods have the potential to create an initial DB state that is shared by many test cases. However, this approach can adjust only inputs and cannot adjust the initial DB

state, hence it cannot always provide an appropriate initial DB state and inputs for each test case.

4.3 Proposed Method

To solve the problems of the existing methods, we propose new method for generating an initial DB state that can be shared by multiple test cases by extending the one-by-one method introduced in the previous chapter. As a result, it reduces the number of initial DB states and the total number of DB records compared to the existing approach, which cuts the cost of test execution.

In this section, we first list up the three challenges that must be overcome in rectifying the weaknesses of the existing approach. Second, we present a framework that solves the challenges and generates initial DB states shared by multiple test cases. Finally, we solve each challenge within the framework.

4.3.1 Challenges

There are three challenges to achieve generating single initial DB states that are shared by multiple test cases.

1. How to divide the test cases into groups; each group member uses the same initial DB state.

2. How to decide the number of records and the alignment of the records in each DB table.

3. How to create concrete values of each initial DB state that satisfies the DB schema and returns appropriate numbers of records for each DB search conditions set in the test cases that belong to the same group.

Challenge (1): Ideally, one initial DB state should be shared by all test cases, but this is seldom possible. If n is the total number of test cases, the total combination number of grouping, C, is given by $C = \sum_{i=1}^{h} S(n, i)$. S is Stirling partition number. $S(n, k)$ denotes the number of ways to partition a set of n objects into k non-empty subsets. Hence, C increases exponentially with test case number. Therefore, a creative approach is needed when dividing them into groups.

Challenge (2): After grouping the test cases, the second challenge is how to decide the number of records and the alignment of the records in each DB table. For example, considering the initial DB state in Figure 4.3, the initial DB state is shared by both test

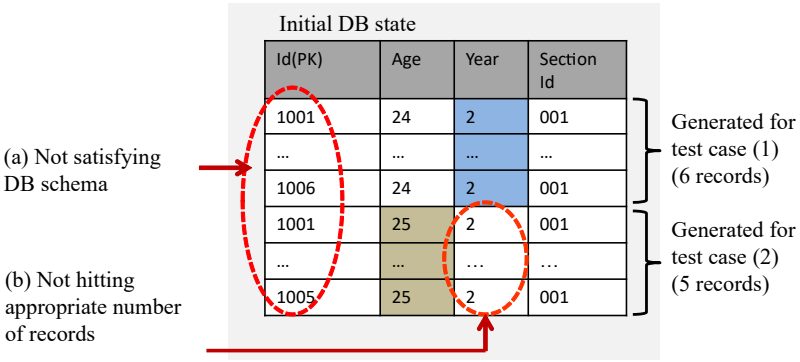


Figure 4.4: Problems When Naively Combining Records

cases (1) and (2), and has six records. However, other initial DB states are appropriate for the two test cases; they may have different numbers of records and different alignments from the initial DB state shown in Figure 4.3. For instance, an initial DB state that has eleven records where the first six records satisfy the DB search condition of test case (1) and the remaining five records satisfy the DB search condition of test case (2). Reducing the number of records, reduces test data size. However, depending on the number(and the alignment) of records, it may become impossible to solve all constraints.

Challenge (3): After deciding the number and alignment of records, the third challenge is how to create the concrete values of the initial DB state that satisfy the DB schema and return appropriate numbers of records for each DB search condition set in the test cases that belong to the same group. It is impossible to naively combine the DB records generated for each test case by the existing approach, because the resulting set of records will likely contain duplicate primary keys, see the example in Figure 4.4(a). There is another problem with naive combination; the number of records returned from the combined initial DB state may differ from the number expected for each test case. Figure 4.4(b) shows that test case(1) now returns eleven records. Therefore, to make an initial DB state shared by multiple test cases satisfy the DB schema and return the appropriate number of records for DB search conditions in each test case, a creative approach is needed when generating constraints for the initial DB states.

4.3.2 Overview of Proposed Method

To solve the three challenges discussed in the previous section and generate an initial DB state shared by multiple test cases, we propose a method called DDBGenMT shown in Figure 4.5.

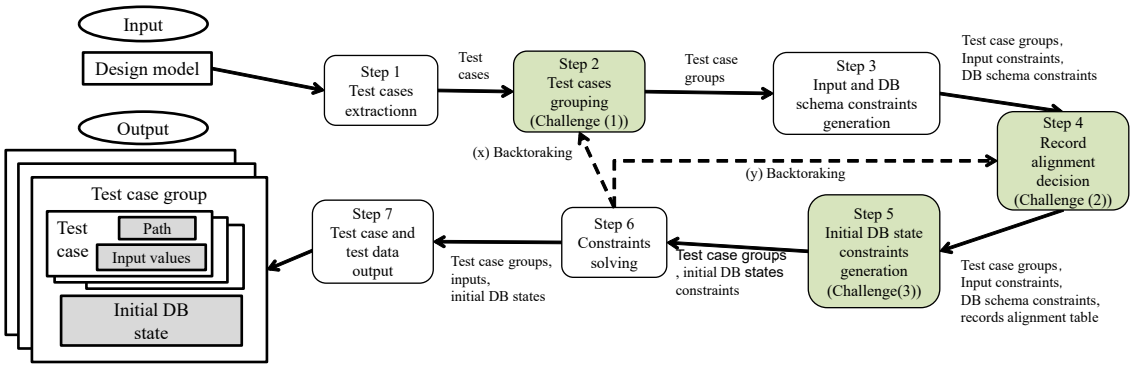


Figure 4.5: Overview of Proposed Method

Table 4.1: Definition of Test Case Group

Index	Definition		
1	<TestCaseGroup>	::=	TestCaseId:String+ <InitialDatabaseStateValue>
2	<TestCase>	::=	PathId:String <InputValue>+

DDBGenMT is an extension of the one-by-one method introduced in the previous chapter. Note that the DDBGenMT do not deal with the constraints of the foreign key and the partial string matching. This is a restriction of the proposed method. Therefore, the DDBGenMT takes a design model with these constraints excluded from another design model shown in the table 3.1 and 3.2 as the input. The output of the method is *test case groups*. Definitions of test case groups and test cases in the DDBGenMT are shown in Table 4.1. A test case group has one initial DB state (Table 4.1(1)) and a test case has only path and inputs (Table 4.1(2)) because an initial DB state is shared by multiple test cases that belong the same test case group. We explain each function in our method below (Figure 4.5).

1. Test case extraction: extracts test cases from the design model (process flows, input definition, and DB schema).

2. Test case grouping: divides the test cases into some test case groups where each group uses the same initial DB state.

3. Input constraint generation: generates the constraints that the inputs need to satisfy, from the input definition and the guard conditions set on the path.

4. Record alignment decision: decides the number and alignment of records.

5. Initial DB state constraint generation: generates the constraints that the initial DB state must satisfy, from the DB search conditions in each test case and the DB schema.

6. Constraint determination: solves for the constraints generated in Step 3 and Step 5 by using a constraint solver.

7. Test case and test data output: outputs the test case groups where each test case group has an initial DB state and has some test cases and inputs corresponding to each test case.

We can use the techniques of the one-by-one method for steps 1, 3, 6, and 7. The other steps 2, 4 and 5, address challenges (1), (2) and (3) described in the previous section respectively.

4.3.3 Step 2: Test Case Grouping

As discussed in Section 4.3.1, the combinatorial number of grouping is very large, and it is difficult to try all combinations of groupings. Therefore, instead of searching for an optimal solution, we adopt an approach to obtain a practically effective local solution. We set a policy to ensure that the number of trials is not so large even in the worst case, and that an appropriate initial DB state is obtained as a precondition for all test cases.

Based on the policy, our proposed algorithm first tries to generate an initial DB state that can be shared by all test cases (by one test case group). If no solutions can be obtained by solving constraints which the initial DB states must satisfy in step 6, it performs a backtracking as shown in Fig. 4.5(x) and divides the test case group into half groups. It repeats this procedure recursively, as shown in Figure 4.6. If the constraints which the initial database states of the test cases in the same group must satisfy contradict each other, no solution can be obtained. To split the test case group in half aim to resolve the contradiction. As the number of divisions increases, it becomes easier to obtain a solution. Also, the number of times this algorithm tries to solve the constraints on the initial DB state is $\mathbb{O}(N \log N)$ at worst cases for N test cases. Therefore, the amount of calculation does not increase explosively as the number of test cases increases. Moreover, even if an initial DB state shared by multiple test cases cannot be generated, one appropriate initial DB state for each test case is eventually generated like one-by-one methods. Therefore, our algorithm ensure to obtain all the initial DB states required for each test case.

This algorithm divides the test cases into two groups, one for the first half and the other for the second half in accordance with the order of the given test cases. Therefore,

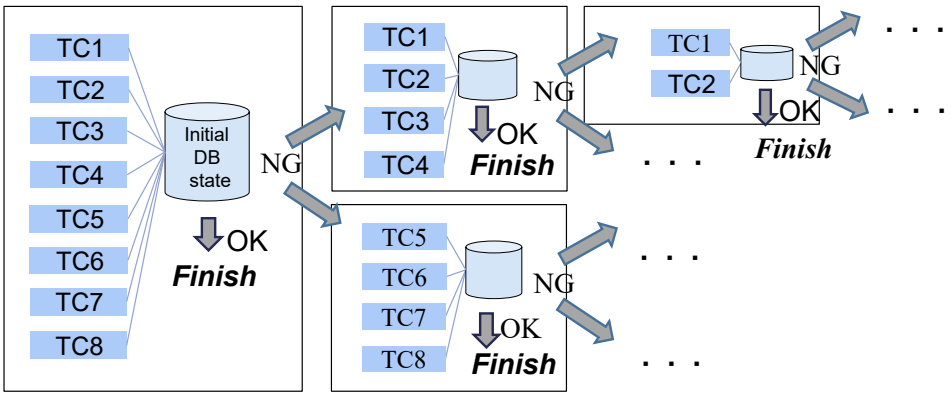


Figure 4.6: Test Cases Grouping

Table 4.2: Calculate the Number of Records From the Result condition

Id	Result condition	Number of records
1	$ResultCount >= C$	C
2	$ResultCount > C$	C+1
3	$ResultCount <= C$	C
4	$ResultCount < C$	C-1
5	$ResultCount == C$	C
6	$ResultCount! = C(C = 0)$	1
7	$ResultCount! = C(C \geq 1)$	C-1

whether a solution can be obtained with only a few backtracking depends on the order of the test cases. This point is discussed in Section 4.4.

4.3.4 Step 4: Record Alignment Decision

This function decides the number of records and the alignments of records in the initial DB state and creates the *record alignment table*.

Before creating the record alignment table, the numbers of each records ($R_i(1 \leq i \leq N)$) required for the N test cases ($TestCase_i(1 \leq i \leq N)$) that access the same initial DB state are determined based on the result condition of the DB search condition of each test case and the concept of the boundary value analysis as shown in Table 4.2.

For example, the test case (1) in Fig. 4.1 is corresponding to the path (1), (2), (4), and (5) of a design model exemplified in Figure 4.7(a). The result condition, which is described in the edge connecting the nodes (4) and (5), of the this test case is $ResultCount >= 6$,

so the number of the records is 6 by calculating based on Table 4.2.

As with the test case grouping, the policy for determining the record arrangement table is to obtain a practically effective localized solution. Based on this policy, our algorithm adopt *minimum records pattern* and *maximum records pattern*, then it tries the two patterns in turn.

- Minimum records pattern: In this pattern, the number of records is minimum, hence it has the advantage of minimizing the size of test data. However, constraint solving is likely to fail because a constraint generated from one DB search condition is highly likely to overlap the constraint generated from another DB search condition. When DB search conditions, where each DB search condition is used in $TestCase_i (1 \leq i \leq N)$, need to return each $R_i (1 \leq i \leq N)$ records, the number of records is $Max(R_1, R_2, .., R_{N-1}, R_N)$ in the minimum records pattern. For example, when creating a record alignment table for the initial DB state shared by the 2 test cases shown in Figure 4.1, the record alignment table shown in Figure 4.8(a) is created in the minimum records pattern. In this case, the value of the initial DB state finally becomes that shown in Fig. 4.3.

- Maximum records pattern: In this pattern, the number of records is maximum, hence it has the disadvantage of maximizing the size of the test data. However, it has the potential to generate solvable constraints because a constraint generated from a DB search condition does not overlap another constraint. The number of records is $R_1 + R_2 + ... + R_{n-1} + R_n$ in this pattern. For example, when creating a record alignment table for the initial DB state shown in Figure 4.1, the record alignment table shown in Figure 4.8(b) is created in the maximum records pattern.

The aim of the maximum records pattern is to make it easier to resolve inconsistencies among the constraints of DB search conditions of the test cases when the minimum records pattern cannot solve them. However, we note that it may not be possible to solve with the maximum records pattern but solve with the minimum records pattern depending on the combination of the DB search conditions and the numbers of DB records. For example, if two search conditions have an inclusive relationship in which one is “five records need satisfy condition A” and the other is “two records need satisfy conditions A and B,” we can obtain a solution satisfies the both conditions only when we use the minimum records pattern instead of the maximum records pattern.

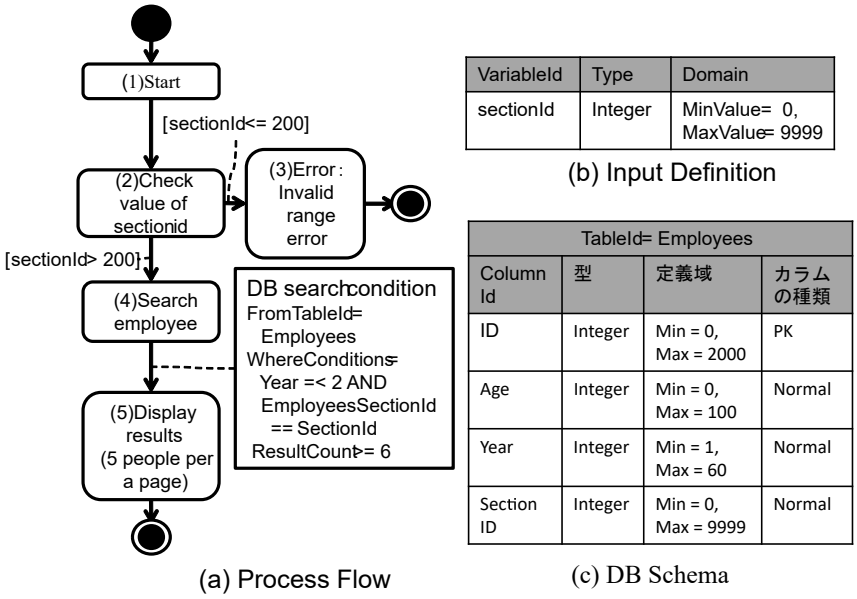


Figure 4.7: Example of Design Model

4.3.5 Step 5: Initial Database State Constraint Generation

If we naively combine the records generated for each test case by one-by-one methods, the problems described in Figure 4.4 occur. Therefore, we need generate the constraints that ensure that the initial DB state satisfies the DB schema and returns appropriate number of records for each DB search condition. We then simultaneously solve the constraints and obtain an appropriate initial DB state shared by multiple test cases as shown in Figure 4.5(6). Our algorithm generates an initial DB state constraint which is a set of constraints that an appropriate initial DB state must satisfy as preconditions for multiple test cases for each table in the DB. The input and output of our algorithm for a table $TableA$ are as follows.

- Input: N test cases $(T_1, ..., T_N)$ refer to $TableA$, a set of the input constraints $\mathbb{C}_{Input}$ and a set of the DB schema constraints $\mathbb{C}_{DBSchema}$ generated in the input and DB schema constraints generation (in Fig. 4.5(3)), and a record alignment table $RATable_A$ generated in the record alignment decision (in Fig. 4.5(4)).
- Output: $\mathbb{C}$ which is a set of the constraints that an appropriate initial DB state should satisfy as the precondition for all test cases that accesses $TableA$.

The pseudo code of the proposed algorithm is shown below.

$\mathbb{C}_{DBTable} \leftarrow \phi$

(a)Minimum records pattern

Employees

Index	Referring test case
0	TestCase01, TestCase02
...	...
4	TestCase01, TestCase02
5	TestCase01

(b)Maximum records pattern

Employees

Index	Referring test case
0	TestCase01
...	...
5	TestCase01
6	TestCase02
...	...
10	TestCase02

Figure 4.8: Examples of Record Alignment Tables

for $i \leftarrow 1$ to N **do**
  $w_i \leftarrow$ WhereClauses of DBSearchCondition of T_i
  for $j \leftarrow 1$ to # of Conditons of w_i **do**
    $w_{ij} \leftarrow j-th$ Conditions in w_i
    $c \leftarrow$ Column of $TableA$ referred by w_{ij}
    $L \leftarrow$ # of records of $TableA$ by referring to $RATable_A$
    $f_1, ..., f_L \leftarrow L$ field variables corresponding to c
    for $k \leftarrow 1$ to L **do**
      $Rec \leftarrow k-th$ record in $TableA$
      $w \leftarrow duplicateof w_{ij}$
      Replace column c in w with f_k
      if W_{ij} refers to Rec in $RATable_A$ **then**
        Add w to $\mathbb{C}_{DBTable}$ //(a)
      else if W_{ij} does NOT refer Rec in $RATable_A$ **then**
        Add logical negation of w to $\mathbb{C}_{DBTable}$ //(b)
      end if
    end for
  end for
end for
$\mathbb{C} \leftarrow \mathbb{C}_{Input} \cup \mathbb{C}_{DBSchema} \cup \mathbb{C}_{DBTable}$

In order to return the appropriate numbers of records for each DB search conditions that access the same table without any excess or shortage, our algorithm creates the constraints to return the required number of records ((a) in the pseudo code) and the other constraints for avoiding to return unnecessary records ((b) in the pseudo code) as the constraints that the initial DB state must satisfy.

For example, when generating the constraints for the initial DB state shared by the two test cases shown in Fig. 4.1 using the minimum records pattern, the constraint set

$\mathbb{C}$ is as follows.

```
Constraints extracted from DB search conditions
E[0].Year <= 2, E[0].Age >= 25,
...
E[4].Year <= 2, E[5].Age >= 25,
E[5].Year <= 2, not(E[5].Age >= 25),
E[0].SectionId = SectionId,
...
```

If the initial DB state returns more than six employee records by the DB search condition of test case (2), it is not appropriate for test case (2). Therefore, our algorithm generate the constraint `not(E[2].Age >= 25)` to make the initial DB states not return more than six employee records.

In step 5, constraint set $\mathbb{C}$ and the other constraint set to satisfy the DB schema (as per the existing approach) are generated. In step 6, simultaneously solving the constraints related to the initial DB state and the other constraints related to inputs which are generated in step 3, we can obtain an appropriate initial DB state shared by multiple test cases that belong to the same test case group and the input values for each test case.

4.4 Evaluation

We evaluate our approach in this section.

4.4.1 Measure

To reduce the cost of test execution, our research aims to reduce the number of initial DB states and the total number of DB records compared to the existing approach. Therefore, as discussed in Section 4.1, we adopt the following two evaluation indexes in the evaluation.

- The number of initial DB states
- The total number of DB records

It is guessed that it depends on the initial order of test cases whether test case grouping works well and can reduce the number of the initial DB states. Therefore, to confirm that, the maximum values (the worst reduction effect) and average values (the average reduction effect) of the number of the initial DB states and the total number of the DB records were also measured when changing the order of the test cases. In addition, we

measured the minimum values of those to confirm whether there is room to increase the reduction effect depending on the order of the test cases.

4.4.2 Procedure

We implemented a prototype tool based on our proposed approach. We used Choco Solver[1] as a constraint solver for solving constraints in step 6.

We selected some of the functions of the industrial-level enterprise systems.

- System X (12 screens): A system that manages schedules and searches data by specified date and time range.
- System Y (11 screens): A web front-end for a network equipment management system that searches data using the IP address as a key.
- System Z (29 screens): A data mining system which searches data under various conditions.

A total of 8 tables were selected from the systems X, Y, and Z, and a total of 87 test cases that had reference access to those tables were selected. We used them for this evaluation.

The prototype tool accepts XML design model files as input. Hence, we first extracted paths from an existing design documents of each system by using an existing method [81]. Second, we manually made XML design models based on the paths, and then added DB search conditions and DB schema, which were extracted from the design document, to the design models. Finally, we input the XML design models to the prototype tool to generate initial DBs and input values. All tasks described to above were performed by a developer experienced in enterprise system creation.

The prototype tool was run on a Windows Vista Ultimate SP2 machine with Intel Core i7(3.0GHz) and 6GB of main memory. The prototype tool can handle the string type, in which the constraints of the equal and the not equal can be handled, and the integer type which are often used in web applications. Therefore, we used integer type as a substitute for enumeration type and time stamp type.

The order of test cases given to the DDBGenMT was the same as one automatically extracted from each process flow of the design models. If there are the tables in which an initial DB state shared by all test cases in a test case group cannot be generated for each table by DDBGenMT, the DDBGenMT was applied 100 times for each table while randomly changing the order of each test cases. Then, we measured how the number of initial DB states and the number of DB records changed. On the other hand, since the

[1] `http://www.emn.fr/z-info/choco-solver/`

Table 4.3: Evaluation Result: Number of Initial DB States and Total Number of DB Records

		One-by-one method			DDBGenMT		
System	DB Table	# of initial DB states	# of DB records	Execution (s) time (s)	# of initial DB states	# of DB records	Execution time (s)
X	X1	3	3	0.249	1	1	0.203
	X2	12	12	0.501	4	10	0.530
	X3	13	49	0.547	5	45	4.236
	X4	15	53	0.624	6	25	0.764
Y	Y1	14	14	0.516	1	1	0.234
Z	Z1	19	10	0.438	1	10	0.218
	Z2	9	4	3.122	1	1	2.684
	Z3	2	1	0.249	1	1	0.187
Total		87	146	6.246	20	94	9.056

results do not depend on the order of each test cases about the other tables in which an initial DB state shared by all test cases can be generated, this measurement was not performed on those tables.

4.4.3 Result

The results of the evaluation are shown in Table 4.3. We confirm that DDBGenMT reduced the number of initial DB states by 23.0% and the total number of DB records by 64.4%, compared to the existing one-by-one method. Table 4.4 shows the numbers related to the backtracking. Specifically, it shows the number of times the test case group was divided in step 2, the number of times the constraint was tried to be solved in step 5 by adopting the minimum records pattern in step 4, and the number of that by adopting the maximum records pattern (it means that the minimum records pattern is failed) in step 4 for each table.

For the three DB tables X2, X3, and X4 in the table 4.3, an initial DB state that could be shared by all test cases could not be generated for each table. Therefore, for these three tables, we applied the DDBGenMT 100 times for each of them by arranging each test cases in random order. The result is shown in table 4.5.

4.4.4 Discussion

Completeness

Even though our method cannot generate an initial DB state shared by multiple test cases, it can eventually generate an appropriate initial DB state for each test case like

Table 4.4: Evaluation Result: Number of Backtrackings

Table	# of division	# of minimum records pattern	# of maximum records pattern
X1	0	1	0
X2	3	1	3
X3	4	4	1
X4	5	6	0
Y1	0	1	0
Z1	0	0	1
Z2	0	1	0
Z3	0	1	0
Total	12	15	5

Table 4.5: Evaluation Result: Shuffled Test Cases

DB Table	DB states	# of initial	# of records	# of divisions	# of minimum records pattern	# of maximum records pattern
X2	Average	3.59	8.42	2.59	1.81	1.78
	Maximum	5	12	4	5	3
	Minimum	2	4	1	1	0
X3	Average	5.33	43.17	4.33	5.12	0.21
	Maximum	8	48	7	8	1
	Minimum	2	22	1	2	0
X4	Average	7.24	44.29	6.24	7.12	0.12
	Maximum	10	50	9	10	1
	Minimum	5	24	4	5	0

one-by-one methods. Therefore, our method was able to generate all the initial DB states for each test case.

Effectiveness of Minimum and Maximum Records Pattern

As shown in Tables 4.4 and 4.5, most of the initial DB states were generated using the minimum records pattern. Since a DB search condition often uses inputs given by the user as parameters, the solution space of the inputs and the initial DB states for a test case group is often large enough. This means that a method using our approach can obtain solutions by adjusting the inputs, as shown by the example in Figure 4.9, in which the degree of freedom of inputs is high, and each input has a different value.

A method using our approach cannot always obtain a solution with the minimum records pattern. As shown in Figure 4.10, if there are constraints "There are two records satisfying `TableA.UPDATE_TIME=UPDATE` in `TableA`" and "There is one record satisfying `TableA.UPDATE_TIME!=UPDATE` in `TableA`," they can be satisfied by using the maximum

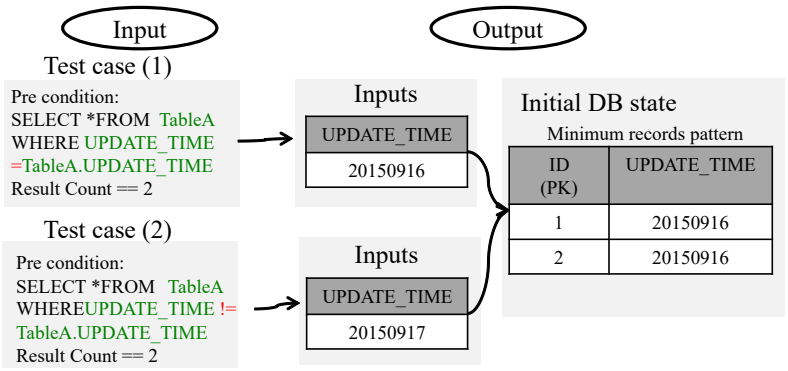


Figure 4.9: Example of Using Minimum Records Pattern to Obtain Solution

records pattern rather than the minimum records pattern.

In addition, when the input values of each test case are constants, the constraints must be solved using the maximum record pattern. For example, if the `UPDATE_TIME` inputs in the two test cases in Fig. 4.9 have the same constant value such as `20150101` (which is not an input), no solution is possible with the minimum records pattern.

As mentioned above, there are cases in which the constraints cannot be solved using the minimum record pattern, so the two-stage method of first trying the minimum record pattern and then the maximum record pattern was effective.

Our method failed to generate an initial DB state that satisfied the preconditions of all test cases for DB tables X2, X3, and X4 in Table 4.3 using either the minimum or maximum records patterns. The reason for this was that there were conflicting constraints, such as "There is one record satisfying condition A" and "There is no record satisfying condition A." An example of this is shown in Figure 4.11. Therefore, to ensure that Step 2 works well, test cases with mutually contradictory constraints, like those in Fig. 4.11, should not be placed in the same group.

Execution time

Since our method does not use a brute-force search in steps 2 and 4, the execution time for test data generation was about 1.5 times that of the existing method. In addition, in cases where the number of initial DB states could be greatly reduced such as the tables in system Z, so DDBGenMT generated test data faster than the existing method. This is because backtracking was rarely necessary in these cases because our method was able to generate an initial DB state shared by all test cases for each table.

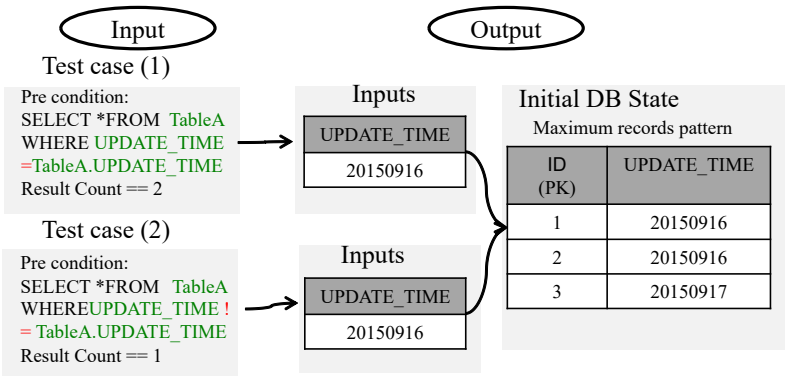


Figure 4.10: Example of Using Maximum Records Pattern Instead of Minimum Pattern to Obtain Solution

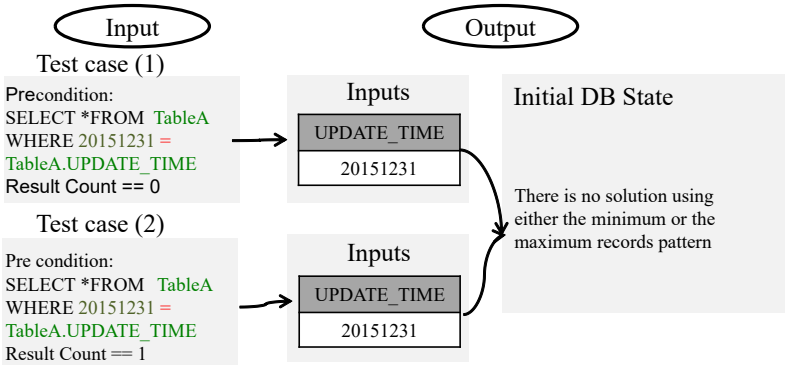


Figure 4.11: Example of Inability to Obtain Solution with Either Minimum or Maximum Records Pattern

Order of Test Cases

As shown in Table 4.5, in the worst case, the number of initial DB states was always reduced by using DDBGenMT, but the number of DB records was not. For the X2, X3, and X4 DB tables, in the average case, DDBGenMT reduced the number of initial DB states by 29.9%, 41.0%, and 48.3%, respectively, and the total number of DB records by 60.1%, 88.0%, and 83.4% respectively. This means that although DDBGenMT was affected by the order of test cases, it nevertheless often reduced the number of initial DB states and on average reduced the total number of DB records.

In practical application of DDBGenMT, better solutions could be obtained by randomly changing the order of the test cases, as long as there is sufficient time to generate test data.

4.4.5 Future Work

We demonstrated the effectiveness of our method in this evaluation. However, as can be seen from the minimum values in Table 4.5, the results in Table 4.3 are not optimal solutions. Therefore, there is room for improvement. For example, the number of initial DB states in the X2 table is four, as shown in Table 4.3, but it can be reduced to at least two, as shown in Table 4.5. Searching for solutions in steps 2 and 4 by brute force or rearranging the test cases randomly many times is time consuming, so more efficient methods are needed to search for solutions. For example, if test cases referring to different columns of the same table were placed in the same test case group in step 2, the conflicts between constraints in step 5 would be reduced. Similarly, an initial DB state with a smaller number of records could be generated by trying out record patterns that are as close as possible to the minimum record pattern while taking care not to overlap the constraints in step 4.

4.5 Chapter Summary

In this chapter, we have proposed a method named DDBGenMT for generating initial DB states that are shared by multiple test cases, and listed the three challenges (test case grouping, DB record arrangement, and initial DB state constraint generation). We have also given simple and reasonable solutions to each challenge. Moreover, using three industrial-level enterprise systems as case studies, we have confirmed that our approach reduces the number of initial DB states by 23%, and the total number of DB records by 64% compared to a one-by-one method.

Chapter 5

Region-based Essential Differences Detection

In this chapter, we describe a method enables the tester to confirm test result efficiently even when there are changes that affect the entire screen based on image-based VRT.

5.1 Introduction

In this section, we explain the problem of the existing image-based VRT systems with a motivating example. As an example of applying VRT, let us consider the login screen of an authentication system for some imaginary application. Figure 5.1 presents an example of differences detection by an image-based VRT system. Figures 5.1 (a) and (b) are screenshot images before and after changes to the application, respectively, while (c) is a screenshot image in which differences are displayed. The VRT system compares the pixels in (a) and (b) at the same absolute coordinate, where the origin of the coordinates is the upper-left corner. If they are the same, the system displays the pixel in grayscale in the differences image; otherwise the system displays it in red. By looking at the differences image, the tester can see at a glance that the "Sign in" button has somehow vanished in the new version of the application. In this way, image-based VRT enables the tester to recognize differences visually and, as a result, efficiently.

Unfortunately, image-based VRT systems can be problematic; if there are changes that affect the *entire* screen, it is difficult for the tester to identify the *essential differences* easily. Consider the case in which the screen design was changed by adding the header "ABCDE Portal Site" in the authentication system described above. This design change resulted in the entire (unchanged) content of the application screen being moved downward, as illustrated in Fig. 5.2. As a result, quite a large number of unessential

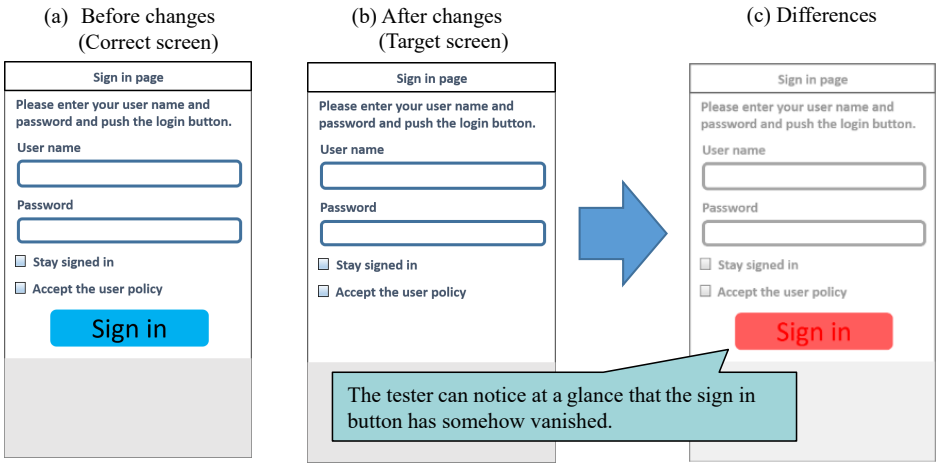


Figure 5.1: Using Existing Image-based VRT to Compare Two Screens in Pixel Units.

"differences" are detected and displayed on the differences screen. The essential differences are difficult to identify because they are buried within a large number of detected differences. There are three main reasons for this problem.

- Screen elements are added or deleted in the new version in accordance with changes in functionality, screen design, and so forth, as exemplified by the case shown in Fig. 5.2.
- There is a region of the screen in which variable-sized elements such as advertisements and the latest news are displayed. We call such a region a *dynamic region* hereafter.
- A bug in the screen element layout results in element misalignment and/or disappearance.

In these cases, it is difficult for the tester to find the essential differences by using an existing VRT system. Thus, the tester must examine the two corresponding screens carefully but is apt to overlook essential differences.

To resolve this problem, we have developed a method for making image-based VRT systems effective even in such cases. The proposed method enables the tester to compare two corresponding screens and efficiently find the essential differences.

The contributions of our proposed method can be summarized as follows.

- We present an image-based VRT method and its prototype system named ReBDiff (Region-based Differences detector) that enables testers to efficiently find essential

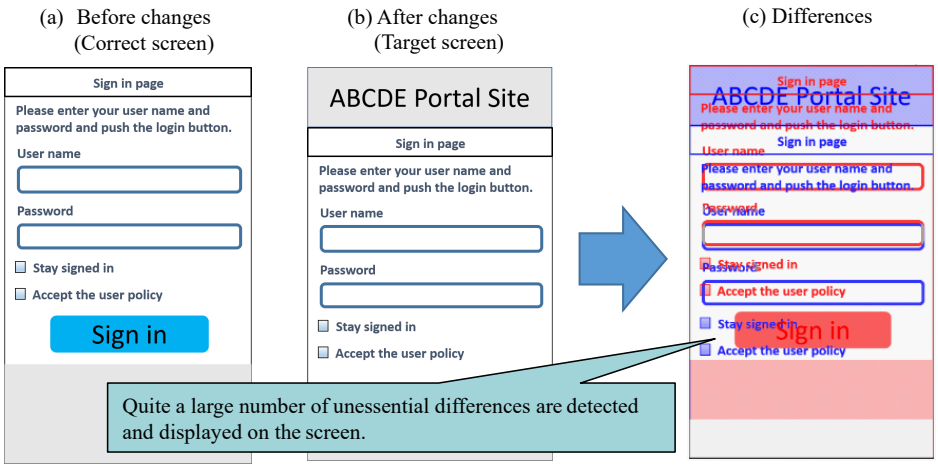


Figure 5.2: Problems in Comparing Two Screens in Pixel Units with Existing Image-based VRT.

differences between before and after screenshot images of an application that has been changed. It divides each image into multiple regions and makes appropriate matchings between the corresponding regions of the two images, and detects such a difference as a shift, an alteration, and an addition.

- We explain how ReBDiff can provide suitable views on the basis of the detected difference types and detailed information about them. By observing these views, the tester can find essential differences between two corresponding screens efficiently even when there are changes that affect the entire screen such as parallel movements of screen elements.

- We describe the experiments we conducted that used websites for both PCs and smartphones, and an Electron application. The results demonstrate the effectiveness of the proposed method.

Since the implementation of the proposed method is ReBDiff, we use "ReBDiff" both to indicate the method and to indicate the tool.

5.2 Related work

Many studies have been carried out on automating or supporting the judgment of test results [9]. This section overviews related research and tools, focusing on VRT.

5.2.1 Implementation-dependent VRT Systems

First we describe VRT that depends on the specific implementation technologies of the target application.

There are VRT systems that use both application screens and structural information at the same time. For web applications, the method proposed by Hori et al. [30] identified screen elements on the basis of document object model (DOM) tree information and compares two corresponding elements to determine whether the web application had been degraded. For cross-browser testing, WEBDIFF [53] and X-PERT [20] identify those places where failures have occurred by comparing the images and DOM trees of the two application screens to detect presentation failures. Ramler et al. [50] presented a method for detecting presentation failures after the user changed the magnification of the desktop in the Windows environment that utilizes both image information and screen element information.

Other approaches exploit only the structural information of application screens. The snapshot test in the JavaScript testing framework (JEST)[1] provides a function that helps the tester ensure the absence of unexpected breakages in the UI layout on the basis of the serialized information of the displayed structures obtained in the tests. Spenkle et al. [58] presented a method for detecting differences by comparing the HTML structures of two corresponding application screens. Takahashi [59] presented a method for recording the history of API calls used for drawing an application screen and comparing the histories for two corresponding application screens. Alameer et al. [4] presented a method for detecting presentation failures after the locale of the application had been changed. A graph is constructed for each screen on the basis of the positional relationships of the elements in the corresponding DOM tree. The graphs of the correct screen and the target screen are then compared. The HTML elements with a changed appearance or a relative position are regarded as being responsible for the observed problem. Walsh et al. [74] presented a method for extracting a "responsive layout graph (RLG)" from a DOM tree and then comparing two corresponding RLGs to detect any undesired distortion of the layout in responsively designed pages.

Several methods for detecting presentation failures, which typically appear in responsively designed pages, work with only the target screen to be tested. ReDeCheck [73] detects overlapping screen elements on the basis of DOM information. VISER [6] goes even further by investigating overlapping at the pixel level, resulting in a higher precision.

These methods are useful for detecting presentation failures in regression testing, but they depend on the specific implementation technology. This means that multiple

[1] `https://jestjs.io/`

platform-dependent implementations must be prepared to enable them to be used on various platforms such as Android, iOS, and Windows.

5.2.2 Implementation-independent VRT Systems

Several approaches are independent of the implementation technology. They attempt to identify the problem from only the application screen images.

The jsdiff and BlinkDiff image-based VRT tools detect differences between two images in pixel units. Similar approaches were taken in VISOR [33] and by Mahajan and Halfond [41]. These tools and methods focus mainly on comparing old and new versions of an application in regression testing. They are effective when the two images to be compared are almost the same, with only minor differences, as exemplified in Fig. 5.1. They are not effective when there are changes that affect the entire screen, as exemplified in Fig. 5.2. Mahajan and Halfond [42] adjusted their method to absorb small differences at the pixel level, but it is still not effective when the positional shift is more than negligible. Lin et al. [39] presented a method that calculates the similarity between the correct screen and the target screen by using several indices such as a histogram of similarity. If the similarity falls below a certain threshold, the target screen under test is presumed to have problems. Although this method can be used to roughly estimate the similarity between two screens on Android terminals with different resolutions, it cannot localize the differences.

Other approaches do not depend on image-based VRT. Visual GUI testing tools [5] such as Sikuli [15] make use of image recognition techniques. Since they treat matching objects on an application screen as images, they can only be used as long as the target objects are present on the screen. They are aimed at ensuring that images are displayed on the screen as expected; they cannot ensure that the screen elements are properly placed without distortion of the screen's appearance. Bajammal and Mesbah [7] presented a method that analyzed a screenshot image of canvas elements in HTML5, identified every visual object and its attributes, and constructed a layered structure of the visual objects for use in generating suitable assertions for the image. Assertions generated for the image of the correct screen can be applied to the image of the target screen to be tested. Unfortunately, their method is applicable only to the canvas in HTML5.

Image-based VRT is flexible and offers two advantages in particular.

- It can be used as long as screenshot images of the application screens are available. Thus, it does not depend on specific implementation technologies such as the OS and web browser.

- It can be easily combined with test automation tools for practical application because such tools generally provide a functionality for obtaining screenshot images.

We have developed a method for making image-based VRT applicable to situations in which there are changes that affect the entire screen and have therefore expanded the scope of its potential application.

5.3 Proposed method

5.3.1 Scope and Requirements

To summarize the discussion in Section 5.2, there are two use cases for VRT. One use case is comparing two corresponding screen images of the old and new versions of an application in the same environment. For example, the "Sign in" screen of the old version is compared with that of the new version in the Chrome browser. The other use case is testing the same version of the target application in various environments, in which a screen image for one environment is compared with the corresponding one for another environment. This includes cross-browser testing and testing on various Android devices.

The application scope of ReBDiff is the first use case with the aim of applying image-based VRT techniques to regression testing even when there are changes that affect the entire screen, exemplified by the three cases described in Section 5.1. The second use case is outside the application scope of ReBDiff. Please note that our aim is to detect differences between two given images, not to determine whether each detected difference is a bug. We assume that the tester is responsible for making that determination.

The application scope defined for ReBDiff means that there are three requirements for a system that supports the tester in detecting and checking essential differences between two application screens by using image-based VRT.

Requirement 1 Each difference can be detected at a level of granularity that makes it easy for the tester to identify the difference.

Requirement 2 All the regions shifted due to changes that affected the entire screen can be checked together.

Requirement 3 Detected differences are displayed with good visibility.

5.3.2 Features of ReBDiff

ReBDiff has three features in particular.

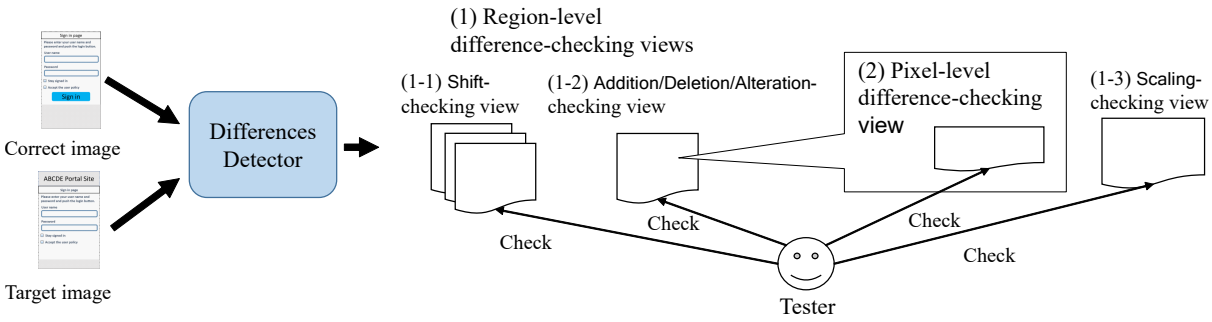


Figure 5.3: Overview of ReBDiff.

First, it detects essential differences in two stages. In the first stage, it roughly detects differences between two corresponding *regions*, one in the correct screen and the other in the target screen under test, and labels each detected difference with one or two *difference type*s among Shift, Addition, Deletion, Alteration, and Scaling. Here, a region is a rectangular section in an image in an application screen. In the second stage, ReBDiff applies an existing image-based VRT method to a pair of corresponding regions. This two-stage process enables the tester to check the differences roughly at the region level (Requirement 1). In addition, the tester can identify parallelly moved regions easily in the first stage (Requirement 2).

Second, ReBDiff groups together regions labeled Shift that have the same direction and amount of movement. Thus, the tester can check these regions together, not one-by-one (Requirement 2). This feature contributes not only to making detected differences visible (Requirement 3) but also to reducing the burden on the tester.

Third, ReBDiff highlights each difference in accordance with its type. Thus, the tester can recognize detected differences with good visibility (Requirement 3).

Figure 5.3 shows an overview of ReBDiff. Given two images, one of the correct screen and one of the target screen, ReBDiff displays two views; one for differences at the region level and the other for differences between corresponding regions in pixel units.

5.3.3 Difference Types

ReBDiff divides the correct and target screens to be tested into regions and detects differences, as presented in Fig. 5.4. For every detected difference, ReBDiff assigns one or two difference types. Currently there are five difference types.

Shift There are highly similar regions in the correct and target screen images, but their positions differ.

Addition The target screen image has a region with no corresponding region in the correct screen image.

Deletion The correct screen image has a region with no corresponding region in the target screen image.

Alteration There are similar regions in the correct and target screen images, but their similarity is not high.

Scaling There are similar regions in the correct and target screen images, but their sizes differ.

Types Shift, Scaling, and Alteration are cases in which there are very similar but not the same regions in both screens. Types Addition and Deletion are exemplified by Region 4' and Region 2 in Fig. 5.4, respectively.

These five difference types should suffice for the following reason. Differences can be divided into two groups: 1) those between corresponding regions in the correct and target screen images and 2) those without corresponding regions in the two screen images. The former can be further classified into Scaling and Alteration on the basis of the similarity level. In addition, if the positions of corresponding regions are not the same, Shift is added. For the latter group, the differences can be further classified into Addition in which a new region is added to the target screen, and Deletion in which a region is deleted from the correct screen.

Precisely speaking, a difference type is assigned to a region pair explained in Section 5.4. For the example shown in Fig. 5.4, region pairs $(null, 4')$ and $(2, null)$, where $null$ means that there is no corresponding region, are associated with Addition and Deletion, respectively. ReBDiff regards regions with high similarity, i.e., greater than a predefined threshold, and with a sufficiently small size difference, i.e., within a predefined threshold, as the "same" and does not detect them as a difference.

The similarity of two regions is calculated on the basis of whether the larger region in height includes an area similar to the smaller region. This is described in more detail in Section 5.4. Thus, there may be cases where similarity is high for regions with different heights. Since such regions need to be checked by the tester, Scaling is added for the regions (a region pair) as a difference type.

Among the five types, Shift, Scaling, and Alteration have additional information on the difference. This information is made explicit by using the following notations.

- Shift (dx, dy): dx and dy are the amounts of movement in the horizontal and vertical directions, respectively.

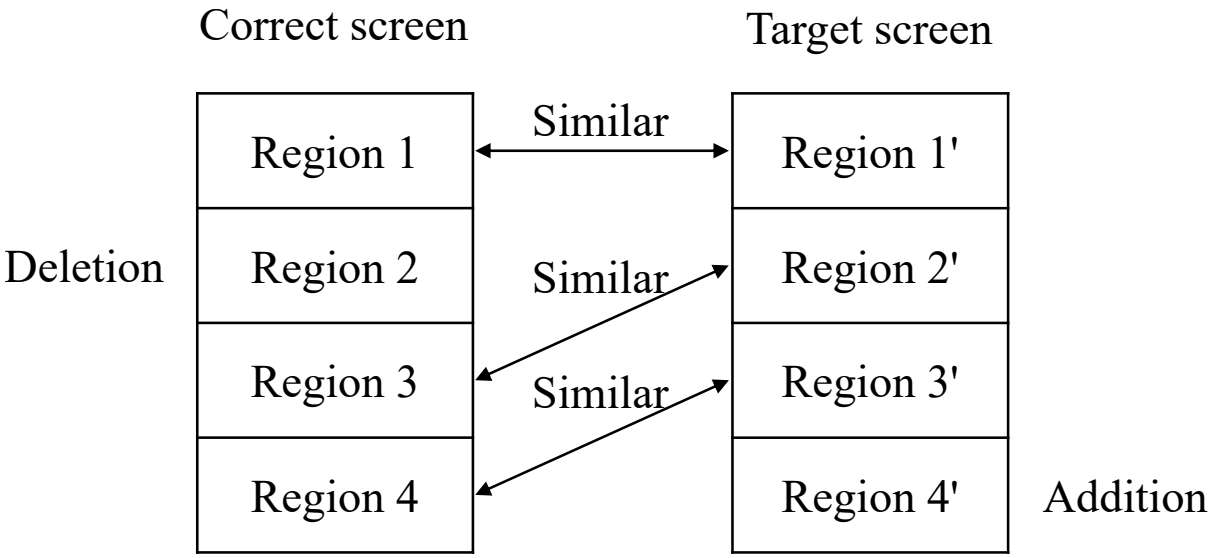


Figure 5.4: Detecting Differences in Region Pairs.

- Scaling (sx, sy): sx and sy are the scaling factors in the horizontal and vertical directions, respectively.
- Alteration d: d represents detailed information on differences at the pixel level.

Please note that two types, namely "Alteration and Shift" or "Scaling and Shift," might be assigned to a detected difference. The details are described in Section 5.4.3.

5.3.4 Difference Checking by Tester

First, the tester checks the Shift differences (Fig. 5.3 (1-1)). The tester can check each group of region pairs with the same direction and amount of movement as a whole by using the *Shift-checking view*. For example, Fig. 5.5 presents an instance of this view when the input screens are those in Fig. 5.2 (1) and (2). In this example, both the correct and target screens are divided into five regions. Since four region pairs except the upper-most one move together vertically a distance of 57 pixels, they are displayed within a red frame. Seeing this view, the tester judges that this difference is not a problem because the movement of these four region pairs is the result of the change in the upper-most part of the screen. In this example, there is only a single group of region pairs. When there are multiple groups, the tester checks them one-by-one by using the Shift-checking view.

After checking the Shift differences, the tester proceeds to check Addition, Deletion, and Alteration differences in the *Addition/Deletion/Alteration-checking view* (Fig. 5.3 (1-2)). For the example in Fig. 5.2, Fig. 5.6 presents the view used for this check, where

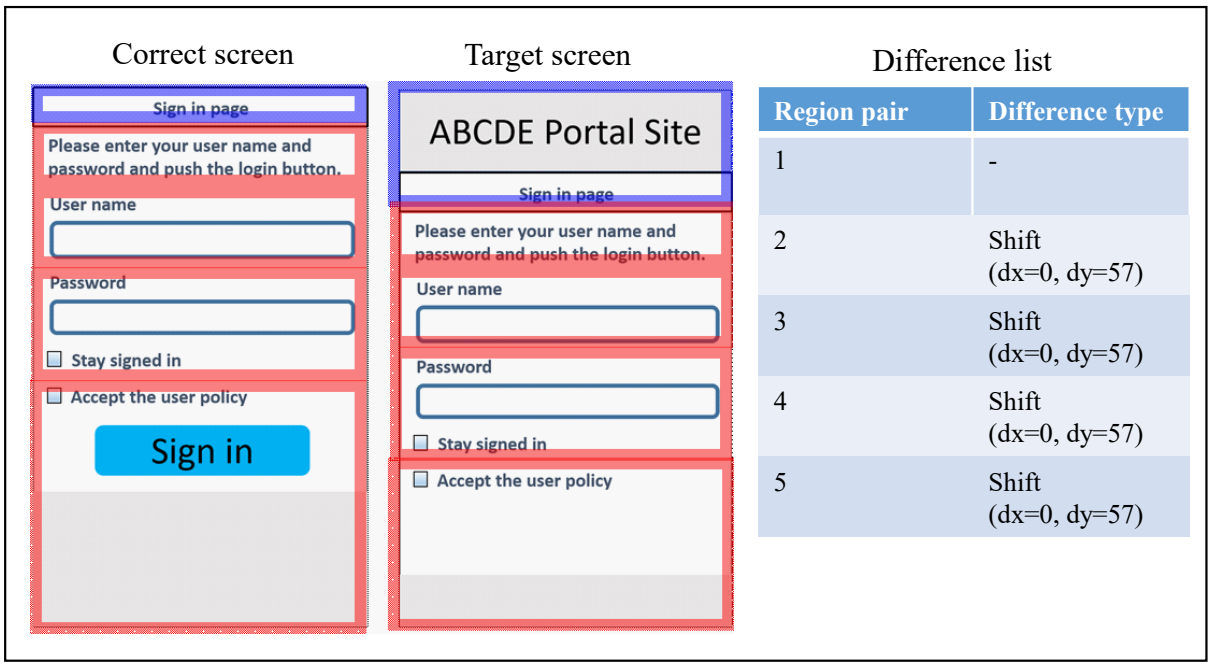


Figure 5.5: Shift-Checking View (Group Identifier is 1).

two differences in Region pairs 1 and 5 are detected. The tester is shown the type of each difference at the right side of the view. When the tester selects a difference in the list (the selected difference is displayed in yellow), the corresponding region is highlighted.

The tester can see the details for an Alteration difference in two ways.

- The tester can compare two regions in pixel units by investigating their overlapped image generated by ReBDiff. When the differences are local and not so large, the place where changes occur can be easily recognized.
- The tester can check the differences by visual examination. Although a visual examination is burdensome, ReBDiff reduces the burden because the area of the region to examine is (much) smaller than the entire screen.

For the example in Fig. 5.7, region pair 5, where the "Sign in" button has vanished in the test screen, can be checked by examining the overlapped image. In contrast, region pair 1's change is difficult to grasp by examining the overlapped image, so a visual examination is needed.

Finally the tester checks for Scaling differences by using the *Scaling-checking view* (Fig. 5.3 (1-3)). For each Scaling difference, scaling factors (both horizontal and vertical) in percentage are displayed in the list at the right side of the view.

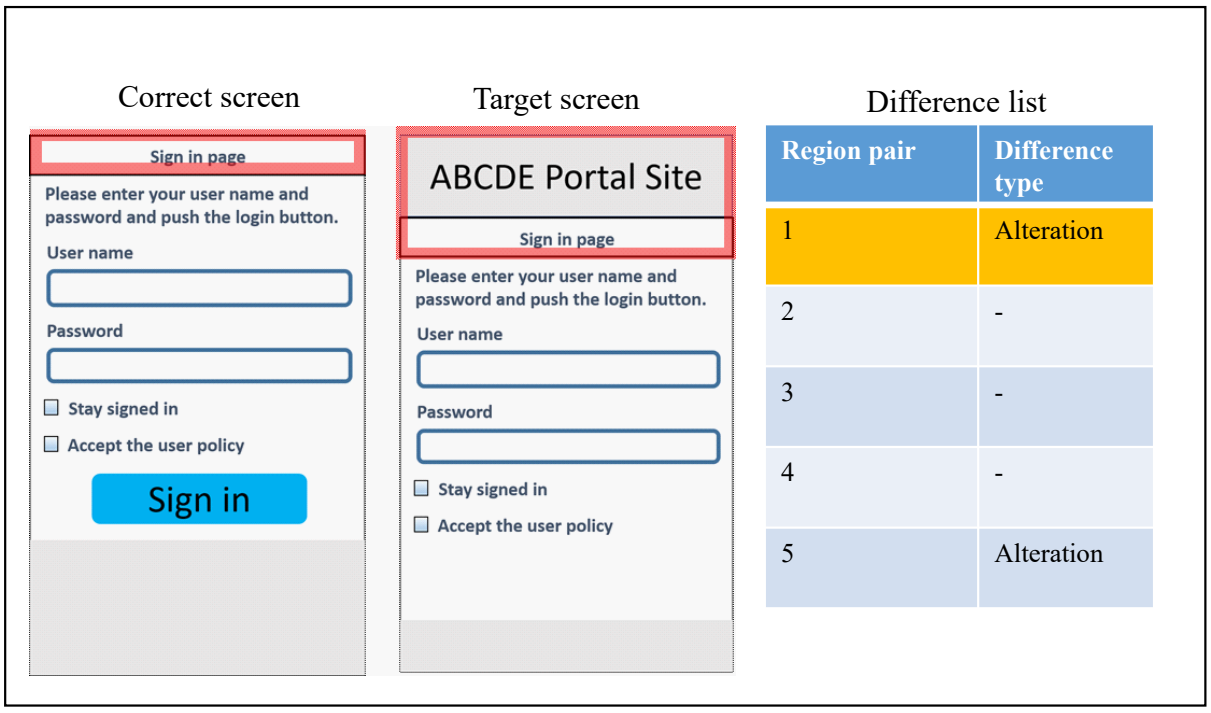


Figure 5.6: Addition/Deletion/Alteration-Checking View.

In this way, comparing differences at the region level makes it possible to roughly check the differences in accordance with the type(s) of each difference.

5.4 Differences Detector

We implemented our differences detector, shown in Fig. 5.3, by utilizing computer vision techniques [29]. Specifically, we used the Python bindings of OpenCV 3.1.0.

Let C be the image of the correct screen and T be the image of the target screen under test. Differences between C and T are detected automatically in three steps.

Step 1 ReBDiff divides C into m regions $c_1, \ldots, c_m$ and T into n regions $t_1, \ldots, t_n$.

Step 2 ReBDiff extracts *region pair*s, each of which consists of a region in C and a region in T that are similar to each other, and creates a list PL of region pairs.

Step 3 ReBDiff assigns one or two difference types to every pair in PL.

Let r, r_1, and r_2 be regions. We assume that $ul(r)$ and $lr(r)$ represent the coordinates of the upper-left corner of r and that of the lower-right corner of r, respectively. In

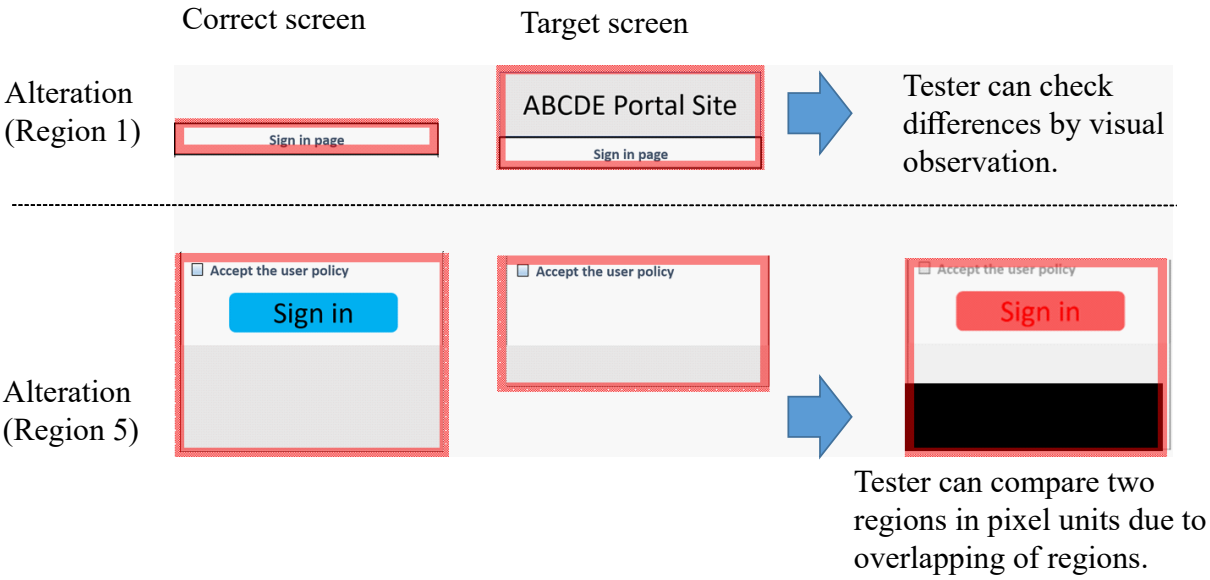


Figure 5.7: Checking for Alteration Differences.

addition, $wd(r)$, $ht(r)$, and $size(r)$ represent the width, height, and size (number of pixels) of r, respectively. We also assume that $sim(r_1, r_2)$ is the similarity of r_1 and r_2. We define the similarity between two regions as the result of template matching [29] on one region using the other region as a template image. The range of this similarity value is 0 to 1. Specifically, we calculate $sim(r_1, r_2)$ by using the `cv2.matchTemplate` method, which performs template matching, and the `cv2.minMaxLoc` method, which obtains the maximum value of the similarity from the template matching results.

Hereafter, we will explain each step.

5.4.1 Step 1: Divide Images into Regions

The first step is to divide C and T into regions in accordance with sub-steps 1-1 to 1-4 below. There are two modes for dividing an image into regions. One is *H-mode*, which divides an image horizontally. This mode is used for vertical screen images in which their contents are horizontally arranged such as web pages for mobile devices and screens of Android/iOS applications. The other is *HV-mode*, which divides an image horizontally first and then further divides each divided region vertically. This mode is used for screen images in which the contents are both vertically and horizontally arranged such as web pages for PCs and Windows native applications. The tester can specify which mode to use in accordance with the features of the screen under test.

The procedure for dividing an image into regions comprises four steps.

Sub-step 1-1 ReBDiff applies Canny edge detection [14] to C to detect the edges. Specifically, by using the `cv2.Canny` method, ReBDiff generates a binary image in which pixels representing the edges are white and the other pixels are black. Then, ReBDiff performs line detection in the horizontal direction on C as follows. If there is a row where the number of white pixels, which represent edges, exceeds $wd(C) \times S_L$ in the generated binary image, ReBDiff regards the row as a line. If multiple consecutive rows are regarded as lines, only the middle one is taken and the others are deleted. S_L is a predefined parameter with a value that should be empirically determined so that an appropriate division of images into regions can be obtained. In our experiments, described in Section 5.4, we set S_L to 0.8. Line detection is used to divide C into K_1 regions: $C = r_1^1, \ldots, r_{K_1}^1$.

Sub-step 1-2 Starting from r_1^1, ReBDiff repeatedly concatenates adjacent regions until the area of the concatenated region exceeds a predefined threshold. Let the concatenated region be r_1^2. Then ReBDiff performs the same process starting from the region in which the concatenation had terminated. Repetition of this process until no region remains results in C having K_2 regions: $C = r_1^2, \ldots, r_{K_2}^2$. The threshold is $wd(C) \times S_R$, where S_R is a predefined parameter. The following is the pseudo code for this process.

$limit \leftarrow wd(C) * S_R$;
$h \leftarrow 0;\ from \leftarrow 1;\ j \leftarrow 1;$
for $i \leftarrow 1$ **to** K_1 **do begin**
  $h \leftarrow h + ht(r_i^1)$;
  if $h \geq limit$ **then begin**
    $r_j^2 \leftarrow$ a region where r_{from}^1 to r_i^1 are combined;
    $h \leftarrow 0;\ from \leftarrow i + 1;\ j \leftarrow j + 1;$
  end;
end;
if $from \leq K_1$ **then begin**
  $rest \leftarrow$ a region where r_{from}^1 to $r_{K_1}^1$ are combined;
  $r_{j-1}^2 \leftarrow$ a region where r_{j-1}^2 and $rest$ are combined;
end;
$K_2 \leftarrow j - 1$;

If the value of S_R is too small, many small regions are generated. As a result, it would take much time to calculate the region pairs in Step 2. In addition, the correspondence accuracy for region pairs would be reduced. Therefore, it is

necessary to set an appropriate value of S_R empirically to generate moderately sized regions. In our experiments, we set S_R to 0.1.

Sub-step 1-3 From $r^2_1, \ldots, r^2_{K_2}$, ReBDiff creates $C = r^3_1, \ldots, r^3_{K_3}$ by merging a single-colored region and an adjacent multi-colored region into a single region. This step is necessary because if many single-colored regions are eventually generated in Step 1, it is likely that Step 2 cannot properly associate a region in the correct image with a region in the target image under test. As a result of this merging, C does not contain a single-colored region if the processing image is not single-colored. Pseudo code for this process is as follows.

```
from ← 1; j ← 1;
for i ← 1 to K2 do begin
  rr ← a region where r^2_from to r^2_i are combined;
  if rr is a multi-colored region
    r^3_j ← rr;
    from ← i + 1; j ← j + 1;
  end;
end;
if from ≤ K2 then begin
  rest ← a region where r^2_from to r^2_K2 are combined;
  r^3_{j-1} ← a region where r^3_{j-1} and rest are combined;
end;
K3 ← j − 1;
```

Sub-step 1-4 For H-mode, $r^3_1, \ldots, r^3_{K_3}$ is directly the resulting list of regions. For HV-mode, ReBDiff performs the same process (Sub-steps 1-1 to 1-3) for every r^3_i ($1 \leq i \leq K_3$), where line detection in Step 1-1 is done in the vertical direction.

Following the above steps, ReBDiff obtains a list of regions for C, i.e., $c_1, \ldots, c_m$. ReBDiff performs a similar process for T and obtains $t_1, \ldots, t_n$.

5.4.2 Step 2: Generating List of Region Pairs

The next step is to create a list of region pairs by repeatedly associating a region in C with a region in T one-by-one on the basis of the similarity between the two regions. Since the cost of calculating similarities for all possible pairs is too large, ReBDiff calculates the similarity only for those regions with coordinates close to each other.

Let R_H and R_V be ranges in the horizontal and vertical directions, respectively, used to search the corresponding region, and let S_P be the similarity threshold used to judge whether two regions are pairable. As before, PL is a list of region pairs, which is initially empty.

We define the similarity between two regions as the result of template matching [29] on one region using the other region as a template image. Of the two regions, the one with the smaller area is used as the template image. PL is created using a two-step process.

Step 2-1 For region c in C, ReBDiff selects regions t from T, each of which satisfies three conditions: (1) t exists in a rectangular region for which the upper-left coordinate is $ul(c) - (R_H, R_V)$ and the lower-right coordinate is $lr(c) + (R_H, R_V)$; (2) $wd(c) = wd(t)$; and (3) $sim(c, t) > S_p$. Let t be the region with the highest value of similarity with c among the regions selected from T. If such a t exists, ReBDiff adds a region pair (c, t) to PL and removes t from T; otherwise, ReBDiff adds a region pair $(c, null)$ to PL. This procedure is performed in order from c_1 to c_m.

Step 2-2 For every region t in T that was not selected in Step 2-1, ReBDiff adds a region pair $(null, t)$ to PL.

If the value of S_P is too small, many inappropriate region pairs that do not contain *null* may be created. On the other hand, if the threshold is too large, two regions that should be paired and labeled Scaling or Alteration may not be paired, resulting in many $(c, null)$ and $(null, t)$ region pairs (Deletion and Addition) being generated. Because the former situation is more serious, it is necessary to empirically determine that S_P is sufficiently large, but not too large. In our experiments, we set S_P to 0.5.

5.4.3 Step 3: Assigning Difference Types to Region Pairs

The final step is to assign one or two appropriate difference types to every region pair in PL. Let S_M ($S_M > S_P$) be the threshold for similarity. The following logic is used to assign difference type(s) to every region pair $p = (c, t)$ in PL, where $attach(p, ty)$ assigns ty to p.

if $c = null$ **then begin** $attach(p,$ Addition$)$; **return end**
else if $t = null$ **then begin** $attach(p,$ Deletion$)$; **return end**;
if $ul(c) \neq ul(t)$ **then begin**
  $(dx, dy) \leftarrow ul(t) - ul(c)$;
  $attach(p,$ Shift $(dx, dy))$ **end**;

if $sim(c,t) < S_M$ **then begin**
  $d \leftarrow$ differences between c and t in pixel units;
  $attach(p,$ Alteration $d)$ **end**;
else if $size(c) \neq size(t)$ **then begin**
  $(sx, sy) \leftarrow (wd(t)/wd(c),\ ht(t)/ht(c))$;
  $attach(p,$ Scaling $(sx, sy))$ **end**
return;

Please note that both Shift and Scaling or both Shift and Alteration might be assigned to a pair. In fact, changes that can be regarded as a parallel movement and also regarded as an alteration or a scaling are commonly seen. Even for such cases, the tester can check each difference type assigned to the pair by using the checking view corresponding to the type, as described in Section 5.3.4.

Finally, ReBDiff assigns the same group identifier to all pairs in PL with Shift that have the same movement amounts, i.e., the same (dx, dy) value.

5.5 Experiments

5.5.1 Research Questions

To evaluate the effectiveness of ReBDiff, we conducted experiments to answer two research questions.

RQ1 Can ReBDiff detect all differences between the correct screen and the target screen under test? Is the number of detected differences as small as possible? Is the tester's effort for confirming the detected differences sufficiently small?

RQ2 What are the differences in the discovery rate and confirmation time compared with those for manual confirmation?

5.5.2 Method

To answer RQ1, we conducted two experiments.

The first one used target screens with embedded artificial mutations representing changes. We prepared images of correct screens by taking screenshots of real-world applications. We then created target images by embedding changes (described below) into the correct images.

- Additions, deletions, shifts, and scalings of screen elements. In some cases of shifting, two screen elements became overlapped.

Table 5.1: Target Screens.

Experiment	Target screen	App. type	Screen size	No. of type 1 change(s) and description	No. of type 2 change(s)
1	Am	Mobile	411 × 1327	1: Change in design in upper side	1
	Ap	PC	839 × 928	1: Change in design in upper side	1
	Bm	Mobile	411 × 2061	1: Deletion of logo at top	1
	Bp	PC	1042 × 1813	1: Deletion of logo at top	1
	Cm	Mobile	411 × 5672	1: Deletion of advertisement at top	1
	Cm11	Mobile	411 × 5672	1: Deletion of advertisement at top	10
	Dp	PC	1097 × 4200	1: Deletion of advertisement in upper side	1
	Dp11	PC	1097 × 4200	1: Deletion of advertisement in upper side	10
	Em	Mobile	411 × 4490	2: Deletion of advertisement in upper side and link button in middle of screen	2
2	Xe	Electron	765 × 593	2: Change in UI in upper and bottom side	0

- Slight alterations of screen elements.
- Changes in line feed positions and fonts.

The second experiment used screens in which there were *actual* changes in a real-world application. We prepared screenshot images of corresponding old version and new version screens.

To answer RQ2, we asked four participants to detect differences in two ways, i.e., by visually checking the entire image manually and by using ReBDiff. We then measured the rate of differences discovery and the time required for confirmation. For each manual detection, we prepared an Excel sheet with the correct image and the target image side by side so that the participants were able to perform visual confirmation as efficiently as possible not only by looking at the display but also by referring to the Excel sheet. For each participant, the target screen for ReBDiff confirmation differed from that for manual confirmation to prevent learning effects.

The parameters and thresholds at each step in the differences detection were adjusted by using data from real-world websites for PCs and smartphones (excluding websites related to the target screens used in this experiment) so that differences were properly detected. The parameter and threshold values were $S_L = 0.8$, $S_R = 0.1$, $S_P = 0.5$, and $S_M = 0.97$.

5.5.3 Target Screens

Table 5.1 lists the target screens used in the experiments. Am and Ap are login screens for the Japan Pension Service Nenkin net (mobile and PC versions, respectively). Bm

Table 5.2: Results for RQ1: Number of Detected Differences.

Target screen	No. of detected differences					Difference
screen	Shift	Addition	Deletion	Alteration	Scaling	detection
Am	2	1	1	1	0	2/2
Ap	1	0	0	2	0	2/2
Bm	1	2	2	0	0	2/2
Bp	1	0	0	1	1	2/2
Cm	1	0	1	1	0	2/2
Cm11	2	3	4	7	0	11/11
Dp	1	1	1	1	0	2/2
Dp11	1	1	1	10	0	11/11
Em	2	0	0	4	0	4/4
Xe	1	0	0	2	0	2/2

and Bp are login screens for the Internet banking service of the Japan Post Bank (mobile and PC versions, respectively). Cm and Cm11 are the top pages of NTT East's mobile web service, Dp and Dp11 are the top pages of NTT DOCOMO's "My docomo" service for PC web, and Em is the top page of NTT West's mobile web service. Am and Ap have a rather simple design and a small screen. Dp, Dp11, and Em have more complicated designs and larger screens.

For each target screen, the target image had

- changes that affected the entire screen (*type 1* changes) such as insertion of a logo at the top of the page, and
- changes that did not affect the entire screen (*type 2* changes) such as deletion of the login button.

For Cm11 and Dp11, type 2 changes were embedded as much as possible in the entire screen. There were ten such changes.

Xe is the screen of an Electron application (2 KL), which had been developed at NTT. Its target image had two type 1 changes and no type 2 changes. Since the Xe screen was modified by the addition of a new function to the application, it was necessary to confirm that unexpected changes on the new version's screen did not occur elsewhere.

For all target screen displays, we observed that the problem shown in Fig. 5.2 occurred when using an existing image-based VRT that performed difference detection in pixel units.

Table 5.3: Results for RQ1: Ratio of Area.

Target screen	Shift	Addition Deletion	Alteration (confirmable in pixel units)	Alteration (not confirmable in pixel units)	Scaling
Am	90.3%	5.8%	11.7%	0%	0%
Ap	62.7%	0%	38.6%	23.7%	0%
Bm	93.2%	6.8%	0%	0%	0%
Bp	87.6%	0%	76.4%	0%	12.4%
Cm	99.6%	0.4%	2.3%	0%	0%
Cm11	96.4%	3.6%	13.4%	0%	0%
Dp	92.9%	4.3%	2.8%	0%	0%
Dp11	93.0%	4.3%	23.1%	0%	0%
Em	92.2%	0%	13.1%	0%	0%
Xe	68.2%	0%	0%	56.6%	0%

We obtained screenshot images of PC and mobile web screens on the Chrome browser in Windows 10 by using Full Page Screen Capture, which is a Chrome extension. For the mobile web pages, we used Chrome's developer tool to display the pages with the screen size of the mobile version (we used the size of the Pixel2 XL terminal) and then captured screenshot images. For the Electron application, we obtained screenshots by pushing the Alt and PrintScreen keys, the traditional way to get screenshots in Windows.

In addition, for Cm11 and Dp11, we asked the participants to detect differences in two ways: by using ReBDiff and by manual checking of the entire image.

5.5.4 Results

RQ1

Tables 5.2 and 5.2 present the results of applying ReBDiff to each target screen: the number of detected differences for each difference type and the ratio of the area of the regions in which differences were detected to the total area. Here, the total area is the sum of the area of the correct image and that of the target image under test, which can be regarded as the entire area to be checked in order to detect differences. Since manual checking without ReBDiff requires that the total area be entirely checked, the smaller the area of the regions for which ReBDiff will be used to detect differences, the greater the effectiveness of ReBDiff.

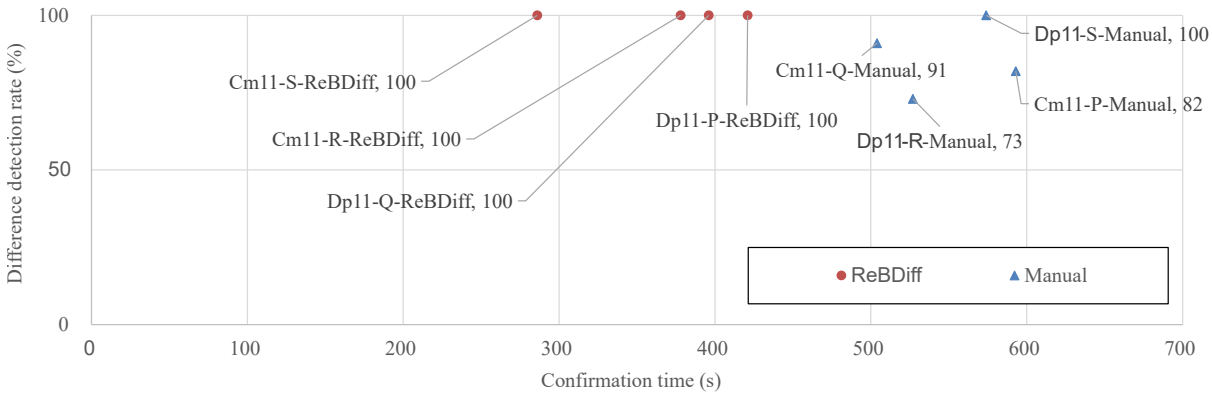


Figure 5.8: Difference Detection Rates and Confirmation Times for RQ2.

For all target screens, ReBDiff detected all (type 1 and type 2) changes. Though the ratios of the area of detected Shift were large, i.e. over 90% for seven cases and at least 62.7%, the number of detected Shift differences in each case was 1 or 2. Therefore, these differences should be confirmable without much effort by the tester.

Examining the Addition and Deletion differences in detail, we see that two regions that should have been paired to form a region pair and detected as a single Alteration difference were detected as an Addition difference and a Deletion difference. This was because they were not similar enough to be paired. In such cases, the tester must expend much effort in checking them because the contents of both the Addition region and the Deletion region must be visually checked to identify the differences between them. However, the ratio of each area for the Addition and Deletion differences was less than 6.8% of the total area, which is not particularly large.

The Alteration differences that were confirmable by comparison in pixel units would not impose a large burden on the tester even though the ratio of the area was as high as 76.4%. This is because ReBDiff provides an overlapping view (Fig. 5.7) that helps the tester compare the two regions.

The Alteration differences for Ap and Xe were impossible to confirm by comparison in pixel units. Though the tester would have to visually check these differences, the burden would be greatly reduced by using ReBDiff because ReBDiff narrowed down the region to be checked (to 23.7% for Ap and 56.6% for Xe). This means that the tester would not need to visually check the entire screen.

RQ2

The difference discovery rate and time required for confirmation are plotted in Fig. 5.8 for both using ReBDiff and manual confirmation. In both cases, there were (the same) four participants, referred to here as P, Q, R, and S. The target screens were Cm11 and Dp11, both of which had one type 1 change and ten type 2 changes. A plotted point labeled "X-Y-ReBDiff" indicates that participant Y confirmed target screen X by using ReBDiff and a plotted point labeled "X-Y-Manual" indicates that participant Y confirmed target screen X manually. When ReBDiff was used, all differences were detected by all participants, whereas when manual confirmation was used, some differences were overlooked. In addition, the times required for confirmation using ReBDiff were much shorter than those using manual confirmation. These results show that ReBDiff improves the difference detection rate and reduces the confirmation time.

Six differences were overlooked in the manual confirmation.

- Differences in font type: 2 instances
- Difference in font size: 1 instance
- Difference in sentence content: 1 instance
- Change in icon position: 1 instance
- Change in logo position: 1 instance

Two participants failed to manually detect differences in font type. Though such differences are difficult to detect visually, they are easily found by ReBDiff by using overlapping and comparing the two regions in pixel units. Some participant also overlooked differences that were relatively easy to find such as a difference in font size and changes in an icon's/logo's position. For such cases, by using ReBDiff, the participants had only to check a limited part of the entire screen and were able to compare an Alteration difference in pixel units. As a result, they found the differences. This demonstrates the effectiveness of using ReBDiff.

5.5.5 Discussion

In the experiments, ReBDiff was able to detect all differences as shown Table 5.2. However, in some cases ReBDiff may overlook a difference. For example, if the area of changes in a region in the target screen is very small relative to the area of the region, this difference might not be detected. Adjusting the threshold values (S_P and S_M) enables the

alteration of regions to be judged more strictly, which reduces such missed detections. Such stricter judgment of equivalences can cause many unessential differences. There is thus a trade-off between reducing the number of oversights and reducing the number of detected unessential differences. An appropriate policy for this might be, in tests where even a few oversights are unacceptable, the equivalence of the two sections should be strictly determined. In tests where time is a priority and a few oversights are acceptable, the equivalence of the two sections should be determined less strictly.

In the experiments, we used screens for PC web and mobile web services and an Electron application with different display sizes and different implementation technologies. The embedded change patterns were created on the basis of interviews with developers who had been mainly developing mobile web applications. Since the effectiveness of the proposed method was demonstrated using these screens and change patterns, the proposed method should be widely applicable. To verify that it can be applied more widely and more generally, it needs to be evaluated using a wide variety of application types, e.g., Android and iOS native applications. Future work also includes interviewing testers at a wider variety of development sites.

As described in Section 5.1, there may be dynamic regions such as those for advertisements and news articles within a screen, and these regions are detected as differences. If many such differences are detected in practical use of ReBDiff the time for checking differences will be longer. A promising method for overcoming this problem is to specify a mask area on the basis of the relative positional relationships of multiple screen elements and use it to remove the dynamic regions from the screen [3].

If the correct screen and the target screen greatly differ, two problems may occur.

- The coordinates of the corresponding regions in the two screens would be too far apart. As a result, a large number of Addition and Deletion differences would be detected because ReBDiff would be unable to create region pairs properly.

- ReBDiff would create an incorrect region pair and detect it as an Alteration difference. If many such Alteration differences are detected, the tester would probably get confused.

A practical solution when more than a certain number of differences are detected for the target screen is to carry out a visual confirmation without using ReBDiff's checking views.

5.6 Conclusion

Our proposed image-based visual regression testing system, ReBDiff, divides each of the images of the two application screens to be compared into multiple regions, makes appropriate matchings between corresponding regions in the two images, and detects differences on the basis of the matchings. By using ReBDiff, the tester can identify essential differences between the two screens efficiently even when there are changes that affect the entire screen, e.g., parallel movements of screen elements. Experiments using screens for PC web and mobile web services and an Electron application demonstrated the effectiveness of the proposed method.

A product [61] incorporating the technology used in ReBDiff is currently being used at NTT group companies. Future work includes improving ReBDiff by reflecting the feedback and comments of actual users.

Chapter 6

Suspend-less Debugging

In this chapter, we describe a method enables the programmer to efficiently debug the interactive and/or realtime parts in db-gui-apps. Our proposed method visualizes the program's internal state which accessing data on DB and/or memory in real time. Since the observation target of proposed method is only the program's internal state, we discuss using programs that accesses the data on-memory instead of on a DB for simplicity in this chapter.

6.1 Introduction

In this section, we introduce a motivating example with a game application, and explain the problems of the existing methods when debugging interactive and/or realtime program.

Suppose that we need to debug the logic of an action game program in which a player character attacks nearby enemy characters. Figure 6.1 presents a fragment of source code for this game written in C#. On the basis of a player's input, this code reduces the health point, which we call HP hereafter, of every enemy within some distance of the player character. An enemy is defeated if its HP becomes zero. This code has a high level of realtimeness because it is executed at a certain interval, say, 30 times a second. This code is bidirectionally interactive because its behaviors change with the input events determined by the player.

Imagine a situation in which the player pushes the key corresponding to `DashButton` on the keyboard, which makes the power of the player character's attack becomes five times stronger than normal. Though the programmer expects an enemy to be defeated if the key corresponding to `AttackButton` is then pushed two or three times, the enemy is not defeated due to a bug. In the rest of this chapter, the keys corresponding to

```
1  public void PlayerAttack(PlayerInput input, Player player,
2     List<Enemy> enemyList)
3  {
4     int damagePoint = player.OffensivePower;
5     if (input.AttackButton)
6     {
7        // player attack strength increases five times
8        if (input.DashButton) damagePoint *= 5;
9        foreach (var enemy in enemyList)
10       {
11          if (Vector3.Distance(player.Position, enemy.Position) <= 5.0)
12          { // enemy is nearby
13             if (!enemy.IsInvincible)
14             { // enemy is not invincible
15                enemy.HealthPoint -= damagePoint;
16                if (enemy.HealthPoint <= 0) enemy.Dead(); // enemy is defeated
17             }
18          }
19          enemy.EndPlayerCollision();
20       }
21    }
22 }
```

Figure 6.1: Code Fragment for Action Game Program.

`DashButton` and `AttackButton` are simply referred to as "dash button" and "attack button," respectively.

There are several possible causes of this bug.

- The value of `input.AttackButton` is false in spite of the player pushing it.
- The power of the player character's attack remains normal because the value of `input.DashButton` is not true.
- `EnemyList` does not include the enemy character that is attacked.
- The result of `Vector3.Distance` is incorrect.
- The value of `enemy.isInvisible` is unexpectedly true.

If we use breakpoint-based debugging, we have to set and then remove many breakpoints to identify the point reached by program execution and to check the values for the variables of interest at that point. Unfortunately, this debugging approach significantly reduces the efficiency of debugging because it forces the program to suspend execution every time it comes to a breakpoint. Moreover, breakpoint-based debugging makes it impossible for the programmer to use some of the features provided by an interactive and/or realtime debuggee program. For the example in Fig. 6.1, it is impossible to push both

the dash button and the attack button at the same time. These problems are common to many interactive and/or realtime programs that handle input events.

We have developed a debugging method that resolves these problems. It visualizes both the information on the execution path and the values of the expressions of interest in realtime. We implemented the proposed method as a debugger named SLDSharp for C# programs. SLDSharp makes it possible to efficiently debug interactive and/or realtime programs such as the one in Fig. 6.1.

The contributions of this work can be summarized as follows.

- We clarify the requirements for a method for debugging interactive and/or realtime programs and then present a debugging method that meets these requirements.
- We present SLDSharp as a specific implementation of the proposed debugging method. Though SLDSharp was implemented for debugging C# programs, its design is applicable to any procedural or object-oriented programming language.
- We describe implementation if SLDSharp by means of code transformation.
- As a case study, we describe the debugging of a game program developed using the Unity game engine that illustrates effectiveness of the proposed method. A demo video of the case study is available online[1].

Our method keeps track of both the currently executing statement in a program and the changes in values of the expressions of interest, and visualizes them in realtime. It enables the programmer to interactively explore possible causes of a bug without having to suspend the program or check a log. Notable features of SLDSharp from the practical point of view are as follows. First, the programmer can select the level (grain) of realtime visualization for the executed parts of a program among the source code file level, method level, and statement level. This enables the programmer to narrow down the parts to be investigated step-by-step. Second, the programmer can specify sections and conditions for visualization. The programmer can thus observe the values of the expressions of interest. Third, SLDSharp can be used to debug multi-threaded programs. This is quite important because many interactive and/or realtime programs use multiple threads.

To implement SLDSharp, we embed code fragments that obtain information needed for debugging in every program statement. Since this embedding is automatically done via program transformation, the programmer does not need to make any changes to the debuggee program in order to use SLDSharp. All the programmer has to do is to invoke

[1] `https://www.youtube.com/watch?v=iI-WG13qx8c`

Table 6.1: Classification of Bugs and Scope of This Work

Bug type		Interactive and/or realtime programs	Batch programs
Functionality	Logical error	Scope of this work	Debuggable using existing debugger
	Runtime error	Debuggable using existing debugger	
Performance		Debuggable using existing profiler	

the program transform and link the necessary runtime library. Thus, this method can be applied to programs written in languages other than C#.

Note that our approach focused on debugging a single process, so debugging a multi-process system is out of scope in this study.

6.2 Related work

Programs that are opposite to interactive and/or realtime programs are *batch programs*, which do their processing without programmer intervention. Typical examples of batch programs are compilers and image processing programs. For both kinds of programs, i.e., interactive and/or realtime programs and batch programs, possible bugs can be classified into three types: functionality bugs that might induce logical errors [56], which is the scope of this work, functionality bugs that might induce runtime errors, and performance bugs that might reduce the speed of program execution unexpectedly.

Table 6.1 summarizes this classification scheme. Here, we review existing debugging methods for these kinds of bugs.

Classical debugging methods

Wong et al. [75] identified four types of classical debugging methods:

- breakpoint-based debugging such as Microsoft Visual Studio Debugger[2] and GNU Debugger (GDB)[3],
- debugging using assertions [52],
- "printf debugging" in which code for printing desired information is inserted, and
- debugging using a profiler such as Microsoft Visual Studio Profiler[4].

[2] `https://msdn.microsoft.com/ja-jp/library/w-indows/desktop/sc65sadd`
[3] `https://www.gnu.org/software/gdb/`
[4] `https://docs.microsoft.com/ja-jp/visualstu-dio/profiling/`

Though classical, these methods are still widely used because of their ease of use. For breakpoint-based debugging, since setting and removing breakpoints are bothersome tasks for the programmer, several researches have tried to ease these burdens. For example, Yia et al. [78] enabled programmable breakpoint setting, Zhang et al. [79] enabled breakpoints to be set automatically without programmer intervention, and Ressia et al. [31] enabled the setting of breakpoints not on lines/statements in source code but on objects by focusing on their state. With any of these methods, however, execution of the debuggee program must be suspended, which causes the problems described in Sect. 6.1. Debugging using assertions and their variants, contracts [26], suspends program execution every time an assertion (or contracts) is violated.

Printf debugging provides a simple way for a programmer to observe the internal states of a program without suspending its execution. However, debugging code must be inserted at many places in the source code. This not only imposes a burden on the programmer but also degrades the maintainability and readability of the program. Although the programmer can separate the debugging code from the main program code by using an aspect-oriented technique [38] or a variation [77], the programmer's burden remains by no means small.

A profiler measures the CPU time consumed by each method or line in the source code on the fly and, when displaying the results, superposes the source code and measured CPU times for ease of visibility. The obtained results are helpful for locating the source of a performance bug and improving program performance, but they are less helpful for identifying a possible source of a logical bug.

Recording execution logs

In addition to the classical methods described above, there are more advanced methods. Some automatically record execution logs of a program for use by the programmer in debugging the program.

ETV [72] records information for the execution paths and displays it visually after the debuggee program finishes execution. JIVE [21] generates an object diagram and sequence diagram from the obtained logs and narrows down the location to be searched by querying to the log information. Hermann et al's [8] target was a reactive program for writing interactive and/or realtime programs. It collects logs by applying patches to the libraries used by the debuggee program and visualizes data flows and timings from the collected logs. Lin et al. [40] reported a method that narrows the range in which bugs may exist through dialogues with the programmer on the basis of automatically collected log information. Maruyama and Terada [43] reported a method that monitors specific lines in the debuggee program and records logs. In the second run of the program, the

Table 6.2: Requirements for Debuggers Applied to Interactive and/or Realtime Program.

		1 Obtain necessary information immediately without suspending	2 High perspective view	3 Low programmer effort	4 Low overhead	5 General applicability
Classical debugging	Breakpoint				✓	✓
	Assertion				✓	✓
	Printf	✓			✓	✓
Recording execution logs			✓	✓		✓
Time travel debugging				✓	✓	
Suspend-less debugging	Tanno2008			✓		✓
	Proposed	✓	✓	✓	✓	✓

method suspends its execution immediately before the occurrences of the target events.

Most of these methods collect logs automatically without suspending the debuggee program and exploit the obtained logs *after* its execution. Visualizing the collected information effectively properly can give the programmer a higher perspective of the entire program for better understanding. Although these methods are useful for debugging, the programmer is unable to see such information as the execution path and variable values *in realtime* and to decide the next input on the basis of such information. As a result, trial and error style debugging is difficult. In addition, since it is impractical to collect *all* information about the target program due to the overhead, the programmer may be unable to find desired or necessary information in the logs.

Time travel debugging

Another approach is to record *all inputs* to a program, not only inputs from the programmer but also those from devices and those generated by programs, e.g., pseudo random numbers. This approach is relatively lightweight because only inputs are recorded.

By collecting and using these inputs, the programmer can replay a program's execution. This approach has been implemented in many programming languages such as C, Java, .NET, and JavaScript (Node.js) [10–12, 35]. The programmer can reproduce any bug encountered at any time and identify the source of the bug by retracing the program's execution. This approach can be regarded as a variation of the log-based one, so these two approaches share the defects described above. In addition, there is a technical hurdle for implementing a mechanism for perfectly replaying execution. For example, multi-threaded programs do not always replays perfectly because of non-deterministic scheduling by the operating system, which cannot be controlled by a user program. Thus, the programs to which this method can be applied are restricted.

Suspend-less debugging for non-preemptive coroutines

The work by Tanno [60] was aimed at solving the problems described above for debugging interactive and/or realtime programs. The targets were game programs written

in a domain specific language (DSL), with the program described as a collection of non-preemptive coroutines. Thus, Tanno's focus was how to debug such coroutine programs. As a result, the work is less general because its target programs are restricted. The programmer is able to choose a coroutine from among coroutines displayed in a list and monitor its behavior. However, the programmer is unable to specify the information to be monitored. In addition, since Tanno's system does not provide a way for obtaining an overview of a debuggee program, it is difficult for the programmer to investigate potential causes of bugs from a higher perspective. The target language is not a widely used one but rather a DSL designed by the author. Thus, the applicability of Tanno's approach is uncertain.

6.3 Proposed debugging method

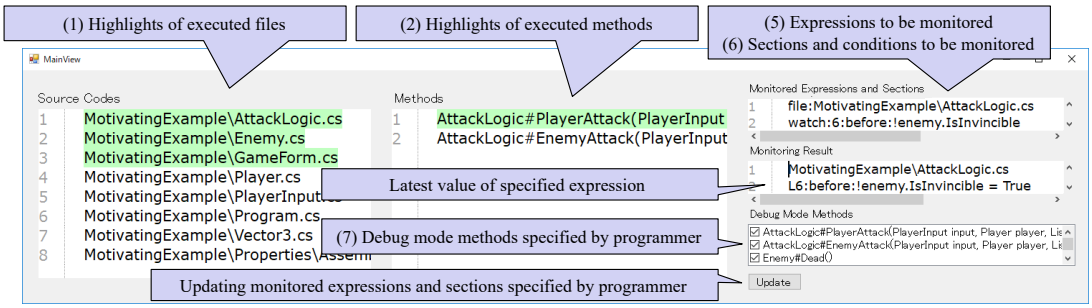
6.3.1 Targets

Our method targets interactive and/or realtime programs in db-gui-apps as shown in Fig. 2.1. Specifically, the targets of the proposed debugging method are functionality bugs that might cause logical errors in the repeated execution process in the flow. This process receives input events from the user, from sensor devices, via the network, and so on. The values received as input events are used to obtain computation results. For example, a specified direction of the movement of the player character from the input device is used to calculate the character's next coordinates. Finally, the results are output onto a display, into memory as internal data, or to somewhere else via a network. In an interactive and/or realtime program, this process occupies most of the execution time. In addition, the behaviors of this process are hard to predict since the program's internal states change moment by moment in accordance with successive arrivals of input events.

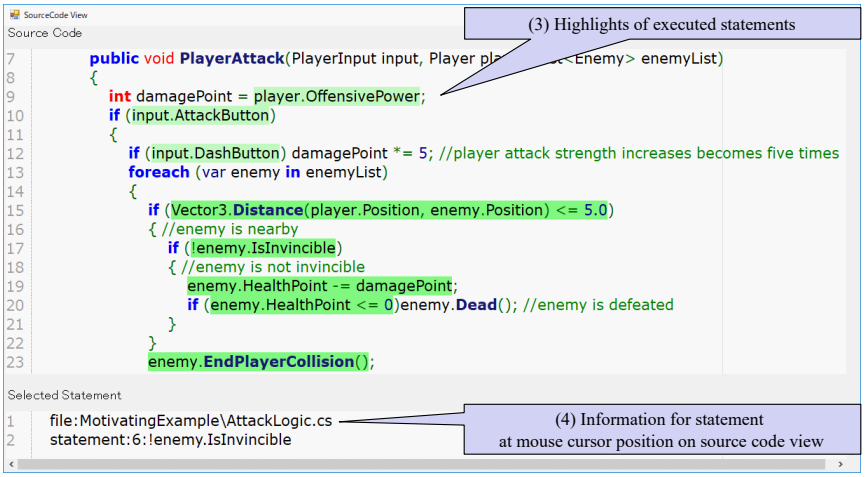
6.3.2 Requirements

From the discussion in the previous section, the requirements for a debugging method for interactive and/or realtime programs that is widely applicable can be summarized as follows.

Requirement 1: The information needed by the programmer for a debuggee program can be obtained immediately without suspending its execution.
Requirement 2: The information obtained can be presented from a high perspective view.



(a) Source file list and method list view



(b) Source code view

Figure 6.2: Views of SLDSharp.

Requirement 3: Obtaining the information does not impose substantial burden on the programmer.
Requirement 4: The added overhead does not degrade the interactiveness and realtimeness of the debuggee program.
Requirement 5: The method is generally applicable to many languages and domains of target applications.

Table 6.2 presents the debugging methods described in Sect. 6.2 in terms of these requirements. We can see that none of the existing methods satisfies *all* the requirements.

6.3.3 Debugging method

The proposed debugging method satisfies *all* the requirements, as shown in the bottom row of Table 6.2. The method has the following features.

Feature 1: The currently executing place, i.e., execution path, in the source code and the values of expressions at a certain interval are presented in realtime. The programmer can thus recognize the internal states of the running program immediately. (Requirement 1)
Feature 2: Information on the execution paths is presented on three levels: file level, function/method level, and statement level. By appropriately switching among these levels, the programmer can grasp the behavior of a debuggee program more clearly. (Requirements 2, 3, and 5)
Feature 3: Detailed monitoring options are provided, such as which sections to monitor, from where to where to monitor, which conditions to monitor, and which expressions, typically variables, to monitor. The programmer thus focus on the information of interest without any noisy information. (Requirements 1 and 5)
Feature 4: Two execution modes are provided: *debug mode* and *normal mode*. The programmer can switch dynamically between them. (Requirement 4)

These features do not restrict the domain of debuggee programs and are applicable to general procedural and object-oriented languages. Features 2 and 3 are not so difficult to implement for many languages and environments by using, for example, the JaCoCo[5] Java code coverage library, code insertion into Java byte codes [18,19], and code insertion by program transformation [51].

6.4 Debugger for C#

Figure 6.2 presents a snapshot of SLDSharp based on the proposed method. SLDSharp has two views, namely *source file list and list view* (Fig. 6.2 (a)) and *source code view* (Fig.6.2 (b)). The main view consists of three parts.

- The left part presents a list of source code file names for the debuggee program.
- The middle part presents a list of C# method names defined in the selected source code file.
- The right part is used by the programmer to specify which sections, conditions, and expressions to monitor. It also displays the values of the monitored variables.

The source code view displays the code fragment of interest.

[5] `https://www.eclemma.org/jacoco/`

SLDSharp continues to display information on program execution, e.g., execution paths and variables values, in these views by means of highlighting every 50 ms. These views give the programmer a higher perspective of program execution and help the programmer efficiently debug the target interactive and/or realtime programs.

In the rest of this section, we explain the characteristic features of SLDSharp by using the example code in Fig. 6.2.

(1) Highlighting source code files

The left part of the main view highlights in realtime the name of the file that has the definition of the method just executed. Highlighting in light green means that the execution has visited this file *once* within a certain amount of time, and highlighting in dark green means *more than once*. By observing the highlighted file names, the programmer is able to grasp which part of the program has been executed. In the example in Fig. 6.2, the code in `AttackLogic.cs`, `Enemy.cs`, and `GameForm.cs` has been executed for deciding whether the player's character should attack an enemy character. If the programmer selects one of the listed files, the middle part of the main view presents a list of methods defined in the source code file. If the programmer further selects one of the methods, the source code view presents its code.

(2) Highlighting methods

The middle part of the main view highlights every method just executed in realtime. For the example in Fig. 6.2, the programmer can verify whether the `PlayerAttack` method in `ActionLogic.cs` has been executed.

(3) Displaying execution paths

In the source code view, the executed statements in the debuggee program are highlighted at regular intervals. By following the highlighted statements as time proceeds, the programmer can grasp the execution paths of the program. In the example in Fig. 6.2, the player attacked an enemy character within a short distance, by pushing only the attack button (without pushing the dash button), resulting in "`enemy.HealthPoint -= damagePoint`" being executed for the enemy character. The programmer can see that the body of the `foreach` statement is highlighted in dark green, meaning that it was executed more than once. If the programmer pushes the dash button, the programmer can immediately verify that "`damagePoint *= 5`" was executed, as shown in Fig. 6.3.

(4) Displaying statement information

SLDSharp assigns a unique identifier to every statement in the debuggee program statically and displays the identifier assigned to the selected statement in a text box. By using these identifiers together with the source code file names, the programmer can specify the sections, conditions, and expressions to be monitored, as explained below.

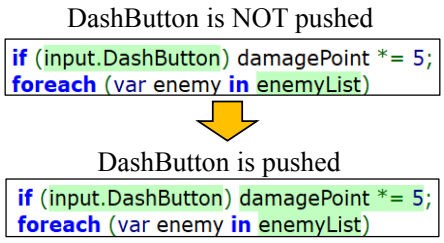


Figure 6.3: Highlighting the Executed Statements.

Table 6.3: Commands for Specifying Sections, Conditions, and Expressions to Monitor.

	Command type	Notation
1	Target source code	`file:`{File path of target source code}
2	Monitored expression	`watch:`{Statement id}`:`{After or before of statement}`:`{Monitored expression}
3	Start of monitoring section	`condstart:`{Statement id}`:`{After or before of statement}`:`{Conditional expression}
4	End of monitoring section	`condend:`{Statement id}`:`{After or before of statement}

(5) Specifying variables to be monitored

The programmer can specify which expressions, typically variables, to monitor by using the commands in Table 6.3. These commands can be used in the right part of the main view. For example, suppose that the programmer wants to monitor changes in the values of `enemy.HealthPoint` just before the statement "`enemy.HealthPoint -= damagePoint`" (Line 13 in Fig. 6.1). To do so, the programmer gives the file name in which this statement is described by using the `file` command "`file:GameForm.cs`." Then the programmer gives the command "`watch:8:before:enemy.HealthPoint`," where "8" is the identifier of the above statement, "`before`" means that the value of the variable should be retrieved just *before* the statement is executed, and "`enemy.HealthPoint`" is the expression that is to be evaluated when the statement is executed. The latest value of this expression is displayed following each repetition of the `foreach` loop.

(6) Specifying sections to be monitored

The programmer can specify which sections within the source code to monitor and the conditions used by SLDSharp to decide whether to monitor. For example, consider the `foreach` loop from Line 7 to Line 18 in Fig. 6.1. If the body of this `foreach` loop is executed at least once, the body will be highlighted. However, from this simple highlighting, the programmer cannot know for which enemy character the body was executed because there might be many enemy characters at the same time on the screen.

SLDSharp enables the programmer to monitor the execution of a sequence of state-

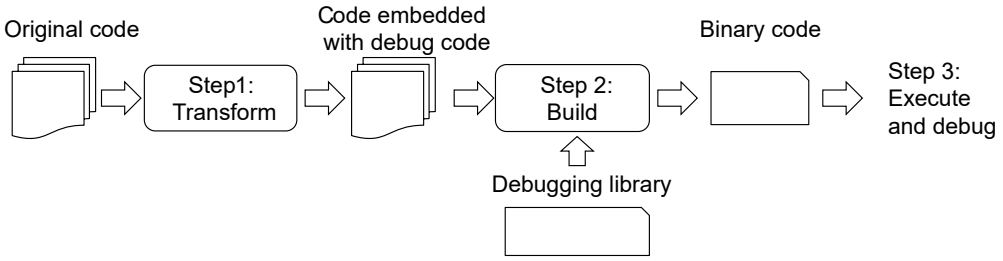


Figure 6.4: Overview of SLDSharp Implementation.

ments for some *specific* condition. For the above example `foreach` statement, if the programmer want to monitor executions of its body for only enemy characters of the type "Slime," the programmer gives the commands "`condstart:5:before:enemy. TypeName ==\"Slime\"`" and "`condend:10:after`" to the first statement (Line 15 in Fig. 6.2) and the last statement (Line 23 in Fig. 6.2) to be monitored, respectively. "`enemy.TypeName== \"Slime\"`" in the `condstart` command specifies the condition under which monitoring should begin. By giving these commands, the programmer can monitor the body of the `foreach` statement only for type Slime enemies and observe the values of expressions if monitoring expressions have already been specified.

For multi-threaded programs, thread identifiers can be used in the condition part of a `condstart` command. Thus, the execution path for a specific thread can be visualized.

The specifications of monitored sections, conditions, and expressions are reflected by pushing the update button in the right part of the main view.

(7) Specifying execution mode

For each method, the programmer can specify whether to execute in debug mode or not. All methods are in debug mode by default.

As explained, a programmer using SLDSharp can debug a program efficiently without suspending its execution and monitoring its behaviors regarding execution paths and variable values.

6.5 Implementation

SLDSharp is implemented in three steps, as shown in Fig. 6.4.

1. The code of the debuggee program is transformed into another program embedded with debugging codes.

2. The transformed code is compiled and linked with our debugging library.

```
1  public void PlayerAttack(PlayerInput input, Player player,
2     List<Enemy> enemyList)
3  {
4     if (_Logger.IsLogging(0))
5        _debug_PlayerAttack(input, player, enemyList);
6     else
7        _original_PlayerAttack(input, player, enemyList);
8  }
9
10 private void _debug_PlayerAttack(PlayerInput input, Player player,
11    List<Enemy> enemyList)
12 {
13    var _l = _Logger.GetLogger(0, 0);
14    int damagePoint = _l.LogFunc(() => player.OffensivePower, 0);
15    if (_l.LogFunc(() => input.AttackButton, 1))
16    {
17       if (_l.LogFunc(() => input.DashButton, 2))
18          //player attack strength increases five times
19          _l.LogFunc(() => damagePoint *= 5, 3);
20       foreach (var enemy in _l.LogFunc(() => enemyList, 4))
21       {
22       ...(omission)...
23          _l.LogAction(() => enemy.EndPlayerCollision(), 10);
24       }
25    }
26 }
```

Figure 6.5: Transformed Code.

3. The binary code is executed, enabling the embedded debugging code to obtain information needed for debugging and the code to be visualized without suspending the program.

Since this embedding is automatically done by program transformation, the programmer does not need to make any changes to the debuggee program to use SLDSharp. Below we describe in detail the first and third steps because they are the major parts of our implementation.

6.5.1 Mechanism of code transformation

SLDSharp embeds debugging code that obtains necessary information for every statement into the code of the debuggee program, thereby generating an instrumented program. We explain code transformation by using the example in Fig. 6.1. Figure 6.5 shows the transformed code, where `_Logger` is a class provided by our debugging library.

To avoid creating extra overhead for obtaining debugging information, SLDSharp has a mechanism for dynamically switching the obtaining of debugging information on and off for each method. In `PlayerAttack` (Line 1 in Fig. 6.5), by checking the return

value of `_Logger.IsLogging(0)` (where `0` is the method identifier for `PlayerAttack`), SLDSharp dynamically switches between the original code and the transformed code without suspending the target program.

The definition of `_original_PlayerAttack` in Fig. 6.5 is the same as that of `PlayerAttack` in Fig. 6.1; `_debug_PlayerAttack` (Line 10 in Fig. 6.5) is the code transformed from the original code for `PlayerAttack`, which is embedded debugging code. The return value of `_Logger.GetLogger` (Line 13 in Fig. 6.5) is an instance of `_Logger` associated with the thread being executed. An instance of `_Logger` is generated for every thread. Each top level statement in the original code is transformed into a method call `_l.LogAction(...)` or `_l.LogFunc(...)` for logging debugging information. `LogAction` and `LogFunc` are logging methods that take a statement, which is a closure of the original top level statement, and the identifier of the statement as their arguments. For example, `player.OffensivePower` in the original code is transformed into `_l.LogFunc(() => player.OffensivePower, 0)` (Line 14 in Fig. 6.5). `LogFunc` is used for the transformation of statements that have return values, and `LogAction` is used for those that do not.

Figure 6.6 shows pseudo code for `LogFunc`. The first argument, `func`, is the closure of the top level statement. `LogFunc` obtains information on execution paths (Line 4) and local variable values to be monitored (Lines 11–13) and determines whether to monitor the specified section on the basis of the conditions given by the programmer (Lines 5–9) before or after an execution of the statement (Line 16). `LogAction` is almost the same as `LogFunc<T>` except that `LogAction` returns no value.

6.5.2 Mechanism of debugging execution

Figure 6.7 illustrates the execution of transformed code for debugging. In this implementation, the debugger controller thread, or *controller thread* for short, runs as an extra thread in a process of the debuggee program. Each thread except the controller thread records, if necessary, logs generated by the embedded code for debugging every time it executes a statement. The controller thread visualizes these logs on the execution paths and the expression values for the programmer.

The controller thread receives designations about the monitored expressions, sections, and conditions from the programmer and stores them in shared memory so that they are accessible from the other threads. As multiple accesses could happen simultaneously, accessing the shared memory needs mutual exclusion.

For execution path information that is frequently accessed by all threads including the controller thread, we used `Interlocked` class, which provides lightweight atomic

```
1  public T LogFunc<T>(Func<T> func, int statementId)
2  {
3     //Debug code before executing statement
4     if(Is logging enable for current thread?) Record that statement executed
5     if(Is condstart(before) exist?){
6        if(Is the condition of condstart true?)
7           Enable logging for current thread
8        else Disable logging for current thread
9     }
10    if(Is condend(before)exist?) Enable logging for current thread
11    if(Is logging enable for current thread?){
12     if(Is watch(before) exist?) Record the values of variables
13    }
14
15    //Execute the original statement
16    T result = func();
17
18    //Debug code after executing statement
19    (...omission...)
20
21    return result;
22 }
```

Figure 6.6: Pseudo Code for `LogFunc`.

operations, to reduce overhead created by mutual exclusion.

Mutual exclusion is also necessary for accesses to the table that stores instances of `Logger` class, each of which corresponds to an executing thread and is retrieved by `Logger.getLogger` in Fig. 6.5. Write accesses to the table happen when a new thread is created. We assume that these write accesses do not happen frequently and thus used `ReaderWriterLock` class, which allows concurrent read accesses but allows write access for only a single thread.

6.6 Case study

The effectiveness of suspend-less debugging using SLDSharp is demonstrated here with a case study using the Tanks tutorial[6] (1.5 KLOC), or Tanks for short. It was developed using the Unity game engine[7], which is widely used for developing game programs. Tanks is a game in which each of the two players, player-0 and player-1, controls a tank and shoot shells at the opponent tank to destroy it. Player-0's tank is red, and player-1's tank is blue. The players use the same keyboard to control their tanks.

We intentionally embedded two faults into the program that resulted in two bugs.

[6] `https://unity3d.com/jp/learn/tutorials/s/tanks-tutorial`

[7] `https://unity3d.com/`

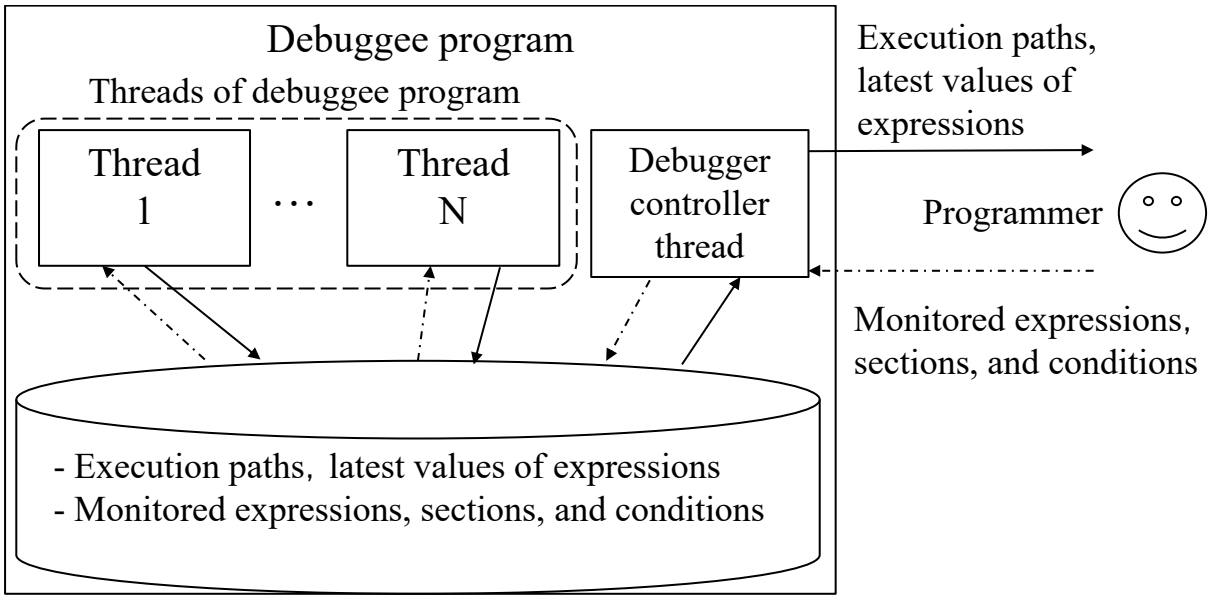


Figure 6.7: Mechanism of Debugging Execution.

- Bug 1: The first bug is that nothing happens when a tank operated by a player shoots a shell and the shell hits the opponent's tank. The expected action is that "when a shell hits a tank, the tank is slightly flipped and damaged, and if several shells hit a tank, the HP of the tank drops to zero and the tank is destroyed"

- Bug 2: The second bug is that "the blue tank does not turn when player-1 pushes the turn button on the keyboard."

Using SLDSharp, we can efficiently identify the causes of these bugs without suspending the program. A demo video of this case study is available online[8].

6.6.1 Debugging First Bug

The detailed process for debugging the first bug is as follows.

As shown in Fig. 6.8 (1), soon after the players start the game, source code file names such as `CameraControl.cs` and `TankMovement.cs` are highlighted. This is because methods related to controlling the orientation and position of the camera, methods related to moving the tanks, and so on are executed at regular intervals, usually 30 to 60 times per second.

[8] `https://www.youtube.com/watch?v=iI-WG13qx8c`

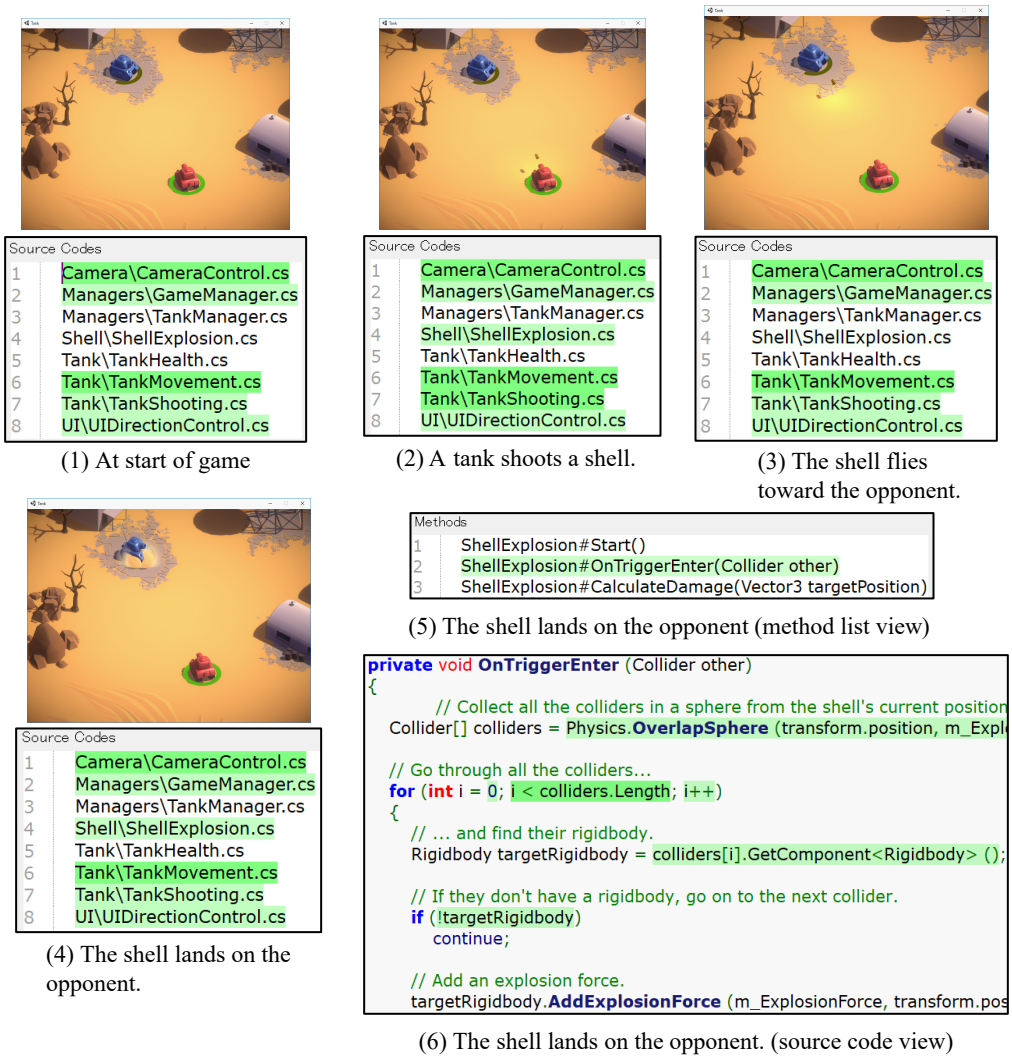


Figure 6.8: Debugging First Bug.

As shown in Fig. 6.8 (2), when a tank shoots a shell, unhighlighted `ShellExplosion.cs` becomes highlighted. Then, as shown in Fig. 6.8 (3), after the shell is fired, the highlight on `ShellExplosion.cs` fades and disappears. In addition, `TankShooting.cs` was weakly highlighted in Fig. 6.8 (1), which means that it was executed only once within a certain amount of time. After the shell is shot, its highlighting becomes strong as shown in Fig. 6.8 (2), which means that it was executed multiple times within a certain amount of time. Thus, we can conclude that `ShellExplosion.cs` and `TankShooting.cs` contain code related to the process for shooting shells.

As shown in Fig. 6.8 (4), `ShellExplosion.cs` becomes highlighted again when the shell lands on the opponent tank. Thus, we can conclude that `ShellExplosion.cs` con-

tains code related to the process for the landing of shells. Since we want to determine why nothing happens when a shell hits the opponent tank, we focus on `ShellExplosion.cs`, which contains the most suspicious code. Next, we investigate which method is executed when a shell lands on the opponent tank.

We fire another shell and check which method is executed when the shell lands on the opponent tank. As shown in Fig. 6.8 (5), when the shell lands on the opponent tank, we observe that the `OnTriggerEnter` method is executed. We thus investigate the behavior of `OnTriggerEnter` in detail.

We use the source code view (Fig. 6.8 (6)) to check which statements in `OnTrigger Enter` are executed. When another shell is fired and lands on the opponent tank, there is one element of `colliders`, and the body of the `for` statement is executed. Since "`target Rigidbody.AddExplosion ...`" is not executed, we can conclude that "`!targetRigidbody`" is true, and the continue statement is executed.

The object assigned to the variable `targetRigidBody` is an instance of `RigidBody` class. Operator `!` is defined on `RigidBody` by operator overloading to be true if the object for collision detection is not set to an instance of `RigidBody`. We expect that the object for collision detection has been appropriately set for each instance of `RigidBody` that corresponds to a tank. Thus, we have identified the cause of the bug — the object for collision detection is not set on the instance of the opponent tank.

6.6.2 Debugging Second Bug

The detailed process for debugging the second bug is as follows.

As shown in Figs. 6.9 (1) and (2), we find that `Turn`, which is an instance method of `TankMovement` class that controls tank turning, is always highlighted regardless of whether the turn button for the blue tank is pushed.

We want to focus on the blue tank only, not on the red tank. In this program, each tank is associated with the controlling player's number: 0 for the red tank and 1 for the blue tank. Thus, as shown in Lines 1–3 in Fig. 6.9 (3), we set the monitored section to be the body of `Turn` (from tstatement 40 to statement 42) and the monitored condition to be `this.m_PlayerNumber == 1`. In this way, we can select which information to monitor by focusing only on the processing of the blue tank.

Though we focus only on the blue tank, `Turn` remains highlighted. This means that `Turn` is still executed for the blue tank but has no effect. To determine the reason, we monitor the values of the variables when `Turn` is next executed.

We can monitor the values of variables such as `turn` and `m_TurnSpeed` by using the monitored expressions. As shown on Line 4 in Fig. 6.9 (3), we use the `watch` command

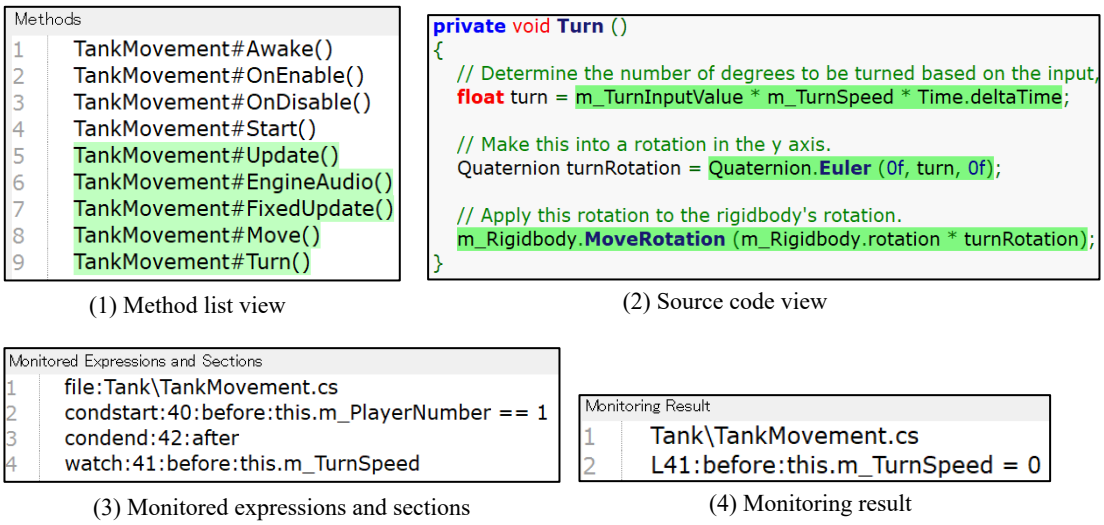


Figure 6.9: Debugging Second Bug.

to monitor the value of `this.m_TurnSpeed`. We then find that the reason for the bug is that `m_TurnSpeed` of the blue tank is always `0` (Fig. 6.9(4)).

As described above, by fully using the features of SLDSharp suspend-less debugging, we can efficiently determine the internal states of the program by giving various inputs without suspending its execution.

6.7 Overhead measurement

To measure the runtime overhead of SLDSharp, We used one GUI program, two game programs, and one numerical calculation program.

1. CalculatorCSharp[9] (2.5 KLOC) is a GUI-based scientific electronic calculator program. We measured the execution time when pushing of the 0 button was repeated 100,000 times.

2. PongCSharp[10] (0.38 KLOC) is a pong game program that accepts player input, computes the movement of the computer player, calculates the movement of the ball, draws the calculation result on the game screen, etc. at regular intervals. We measured the execution time when the calculation and the drawing were executed 1,000 times.

[9] `https://github.com/Cleod9/CalculatorCSharp`
[10] `https://github.com/Cleod9/PongCSharp`

Table 6.4: Overheads for SLDSharp.

Benchmark		Original (ms)	WITH debugger (ms)
1.	CalculatorCSharp	7,360 (100%)	10,344 (141%)
2.	PongCSharp	1,632 (100%)	1,848 (113%)
3.	Tanks	5,747 (100%)	6,097 (106%)
4.	Fib(20)	3,002 (100%)	112,533 (3,749%)

3. Tanks is the game program introduced in Sec. 6.6. It accepts player input, calculates the movement of the tanks and shells they shoot, draws the calculation result on the game screen, etc. at regular intervals. We measured the execution time when the calculation and the drawing were executed 1,000 times.

4. Fib(20) is a numerical calculation program for calculating a Fibonacci number. We measured the execution time when the twentieth Fibonacci number was calculated 5,000 times.

Though numerical calculation programs like Fib are outside the scope of our debugging method, we measured Fib as a reference. In this experiment, SLDSharp only logged and visualized the information on the execution paths and did not monitor the values of expressions. We ran the programs on a Windows PC (Core i7 CPU 2.8 GHz, 14 GB memory, GeForce GTX 1050 Ti, and Windows 10).

As shown in Table 6.4, the overhead for Fib was much larger than for the other three benchmarks. This is because the logging overhead for each top level statement became very high because the calculation of each top level statement was small in the original Fib program. The overheads for PongCSharp and Tanks were smaller than the overhead for CalculatorCSharp because of the large amount of drawing computation performed by the .NET library and Unity engine, which were not monitored by SLDSharp. The overheads of programs targeted by suspend-less debugging (CalculatorCSharp, PongCSharp, and Tanks) were less than 50%, small enough for debugging purposes. Even if the overhead for a debuggee program is not so small, SLDSharp can still be useful by dynamically turning off the debugging mode for the methods not of interest.

6.8 Discussion

The proposed debugging method presents the currently executing place in the source code in realtime. It enables the programmer to interactively explore possible causes of a bug without suspending the program. However, displaying execution paths by highlighting

top level statements might be too coarse for monitoring a statement that has an expression with shortcut evaluation operators.

For example, suppose that the programmer is monitoring the statement "`bool b = X || Y;`" and this statement is highlighted. The programmer can know that this statement has been executed, but cannot know whether `Y` has been evaluated or not only from the highlight of the entire statement. This problem can be solved by monitoring the Boolean expression `Y` using the `watch` command presented in Table 6.3. However, using `watch` command is rather bothersome for the programmer.

Though the proposed method is implemented as a debugger for C# programs by means of a program transformation technique, the technique is not limited to C#. It can be implemented in general procedural and object-oriented languages that provide function closures. However, SLDSharp's implementation on the basis of program transformation imposes the following limitations on the program that uses the reflection mechanism.

- It is impossible to monitor the code that is generated dynamically by the reflection mechanism.
- A program that refers to itself might not behave as expected because its code is transformed.

SLDSharp provides two execution modes: debug mode and normal mode. The programmer can switch the mode dynamically between them to avoid creating extra overhead for obtaining debugging information. However, an execution mode can be set for a method: it cannot be set with finer granularity, e.g., for a statement or for a region of successive statements. Thus, if the programmer want to monitor a method with large logging overhead, there is no way to reduce this overhead. In addition, even if the programmer changes the execution mode of some method on the fly, the new mode is applied from the next invocation of the method, as shown in Fig. 6.5 (line 1-5). Thus, the programmer cannot reflect the mode change of the currently running method without leaving it.

6.9 Chapter Summary

Our proposed *suspend-less debugging* method for debugging logical errors in interactive and/or realtime programs displays information on execution paths and the values of expressions in a debuggee program in realtime without suspending program execution. We implemented this method in SLDSharp, a debugger for C# programs. We demonstrated its effectiveness through a case study using a game program developed using the Unity

game engine. The proposed debugging method was shown to enable a programmer to efficiently debug interactive and/or realtime programs.

Chapter 7

Toward More Efficient Testing

In this study, we dealt with four challenges in test design, test results verification, and fault localization with the aim of automating integration testing for db-gui-apps. The methods introduced in Chapters 3, 4, and 5 have been incorporated into tools used by NTT group companies [71] [64]. This chapter describes the challenges to be faced in achieving more efficient testing at development sites.

7.1 Reducing Cost for Design Model Creation

When introducing the proposed method for initial DB state generation into existing development processes, the cost for design model creation cannot be ignored, especially for development processes in which no design documents are created such as OSS development. A promising approach to solving this problem is to partially create a design model by using specification mining methods such as those proposed by Kurabayashi et al. [36] and Dallmeier et al. [22]. The software specifications are inferred using dynamic analysis, and then a design model is created by applying the missing information to the inferred specifications. For example, input definitions can be partially created using Kurabayashi's method [36].

7.2 Expanding Scope of Initial Database State Generation

The scope of the proposed method for initial DB state generation is limited to checking the behavior of each application logic module when only one screen transition occurs. Moreover, only reference access to a DB and limited queries can be handled, as shown in

Tables 3.1 and 3.2. Therefore, the remaining challenges include how to generate initial DB states in the following test cases.

1. Test cases in which a DB is read from and written to.
2. Test cases for scenario testing in which multiple screen transitions occur.
3. Test cases with complex queries such as subqueries and set operations.

For the first challenge, we generated initial DB states for such test cases [70] [65] by extending the design model introduced in Chapter 3. Source code was generated from a design model for simulating the execution of the software under test, and DSE was applied to the source code to obtain initial DB states in which the DB is read from and written to. For the second challenge, we created initial DB states for scenario testing using the same concept as above [62]. For the third challenge, again on the basis of the same concept, each query in the design model was handled by converting it into source code that simulated its behavior and then applying DSE to the code to get a solution. However, since these methods search for a solution that is an appropriate initial DB state as a precondition of the test case by searching every path in the converted source code, much time may be needed to generate a solution. Therefore, speeding up the search algorithm is an issue to be addressed.

7.3 Reduction of Labor for Test Results Verification

Although ReBDiff reduces the labor needed for test results verification by detecting differences between two screen images automatically, the decision of pass or fail is left to the tester. Although reducing this effort is a remaining task, generating expected results completely automatically is a difficult problem, as mentioned in Section 2.2.3. Therefore, for example, it could be useful to avoid having the tester repeatedly pass judgment. In Adachi et al.'s method [2], for example, the labor needed for test results verification for VRT is reduced by automatically analyzing all the screen comparison results to identify the same differences so that the tester only has to judge one representative screen comparison result. As exemplified by this method, reducing the labor needed to pass judgment is a promising approach.

7.4 Further Debugging Support

Our proposed suspend-less debugging method supports fault localization, enabling the programmer to interactively explore possible causes of a bug without suspending the

program, as shown in the case studies in Chapter 6. As another use of SLDSharp, it can be used in combination with DDBGen (or DDBGenMT) and existing methods [80] [68] [67] [71] to automatically generate a test case. This would enable the programmer to concentrate on identifying the cause of a bug with SLDSharp while the test case is automatically executed, reproducing the bug repeatedly.

To further improve the efficiency of debugging, many methods for automatic fault localization [75] and automatic program repair [44] have been proposed. However, many problems need to be solved before these methods can be practically applied to enterprise systems [46]. Since test cases that can be executed automatically are used to conduct automatic fault localization and automatic program repair, a major problem is how to prepare appropriate comprehensive test cases to improve their accuracy [46] [45]. One possible way to improve their accuracy for db-gui-apps is to use the test cases automatically generated by our study. Another possible way to support debugging is to combine the concept of spectrum-based fault localization (SBFL) [57] and our suspend-less debugging method. This would enable the programmer to narrow down the number of statements that are likely related to the bug in the program. Specifically, a "suspicious score" for each statement could be calculated by comparing some execution paths when the software does not behave as expected with other execution paths when it does behave correctly in accordance with the SBFL. Our debugger could then visualize the suspicious scores overlaid on the source code in real time.

Chapter 8

Conclusion

With the aim of automating the process from test design to test results verification and enabling the cause(s) of each bug detected in testing to be easily identified, we have proposed four methods for solving the major problems in functional testing at the integration level for db-gui-apps. Combining the proposed methods with existing methods/tools could make testing more efficient and contribute to improving the quality, cost, and delivery of software.

The contributions of this dissertation are as follows.

- We discussed the current status and existing methods for testing db-gui-apps and described four major challenges that should be tackled to achieve the goals of this study.

- We proposed using DDBGen for automatically generating on the basis of MBT initial DB states to be entered into a relational DB to support each test case. DDBGen can handle the constraints most frequently used in industrial-level enterprise systems; they include "multiple DB reads," "PK, FK constraints," and "partial string matching." Therefore, high initial DB generation rates were achieved in the evaluations on three industrial-level enterprise systems. In addition, to improve DDBGen, we proposed using DDBGenMT to generate initial DB states, with each one shared by multiple test cases to reduce the number of times the initial DB state must be switched and to reduce the total size of the test data.

- We proposed using an image-based VRT method called ReBDiff. It divides each of the images of the two application screens to be compared into multiple regions, makes appropriate matchings between corresponding regions in the two images, and detects differences on the basis of the matchings. By using ReBDiff, the tester can efficiently identify essential differences between the two screens even when there are

changes that affect the entire screen, e.g., parallel movements of screen elements. Experiments using screens for PC web and mobile web services and an Electron application demonstrated the effectiveness of the proposed method.

- We proposed using a suspend-less debugging method for debugging logical errors in interactive and/or realtime programs in db-gui-apps. It displays information on execution paths and the values of expressions in a debuggee program in real time without suspending program execution. We implemented this method in SLD-Sharp, a debugger for C# programs. We demonstrated its effectiveness through a case study using a game program developed using the Unity game engine. The proposed debugging method was shown to enable a programmer to efficiently debug interactive and/or realtime programs.

- We discussed the remaining major challenges and presented ideas for solving them for testing db-gui-apps toward more efficient testing. Specifically, we addressed four challenges: how to reduce the cost of design model creation, how to expand the scope of the initial DB state generation, how to reduce the labor needed for test results verification, and how to expand debugging support.

Future work will address the remaining challenges and ways to improve the proposed methods in accordance with feedback from development sites. Through such efforts, we will promote more efficient testing and software development.

Bibliography

[1] *IPA/SEC White Paper 2018-2019 on Software Development Projects in Japan (in Japanese)*. IT Knowledge Center on emerging tech trends, Information-Technology Promotion Agency, 2018.

[2] Y. Adachi, H. Tanno, and Y. Yoshimura. Reducing Redundant Checking for Visual Regression Testing. In *25th Asia-Pacific Software Engineering Conference, APSEC 2018*, pages 721–722, 2018.

[3] Y. Adachi, H. Tanno, and Y. Yoshimura. Masking Dynamic Content Areas Based on Positional Relationship of Screen Elements for Visual Regression Testing. (in Japanese). *JSSST Computer Software*, 36(4):53–59, 2019.

[4] A. Alameer, S. Mahajan, and W. G. J. Halfond. Detecting and Localizing Internationalization Presentation Failures in Web Applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 202–212, 2016.

[5] E. Alegroth, R. Feldt, and L. Ryrholm. Visual GUI Testing in Practice : Challenges, Problems and Limitations. *Journal of Empirical Software Engineering*, 20(3):694–744, 2015.

[6] I. Althomali, G. M. Kapfhammer, and P. McMinn. Automatic visual verification of layout failures in responsively designed web pages. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019*, pages 183–193, 2019.

[7] M. Bajammal and A. Mesbah. Web Canvas Testing Through Visual Inference. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018*, pages 193–203, 2018.

[8] H. Banken, E. Meijer, and G. Gousios. Debugging Data Flows in Reactive Programs. In *40th IEEE/ACM International Conference on Software Engineering, ICSE 2018*, pages 752–763, 2018.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[10] E. T. Barr and M. Marron. Tardis: Affordable Time-travel Debugging in Managed Runtimes. In *2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014*, pages 67–82, 2014.

[11] E. T. Barr, M. Marron, E. Maurer, D. Moseley, and G. Seth. Time-travel Debugging for JavaScript/Node.Js. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, FSE 2016, pages 1003–1007, 2016.

[12] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight Recording to Reproduce Field Failures. In *35th International Conference on Software Engineering, ICSE 2013*, pages 362–371, 2013.

[13] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman. On the Dichotomy of Debugging Behavior Among Programmers. In *40th IEEE/ACM International Conference on Software Engineering, ICSE 2018*, pages 572–583, 2018.

[14] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[15] T. Chang, T. Yeh, and R. C. Miller. GUI Testing Using Computer Vision. In *28th International Conference on Human Factors in Computing Systems, SIGCHI 2010*, pages 1535–1544, 2010.

[16] D. Chays, S. Dan, P. Frankl, F. Vokolos, and E. Weber. A Framework for Testing Database Applications. volume 25, pages 147–157, 09 2000.

[17] S. Chen, A. Ailamaki, M. Athanassoulis, P. Gibbons, R. Johnson, I. Pandis, and R. Stoica. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Record*, 39(4), 2010.

[18] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *International Business Machines Corp*, page urlhttp://www.javassist, 1998.

[19] S. Chiba and M. Nishizawa. An Easy-to-use Toolkit for Efficient Java Bytecode Translators. In *2nd International Conference on Generative Programming and Component Engineering, GPCE 2003*, pages 364–376, 2003.

[20] S. R. Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. In *35th International Conference on Software Engineering, ICSE 2013*, pages 702–711, 2013.

[21] J. K. Czyz and B. Jayaraman. Declarative and Visual Debugging in Eclipse. In *2007 OOPSLA Workshop on Eclipse Technology eXchange, eclipse 2007*, pages 31–35, 2007.

[22] V. Dallmeier, B. Pohl, M. Burger, M. Mirold, and A. Zeller. WebMate: Web Application Test Generation in the Real World. In *Seventh IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2014*, pages 413–418, 2014.

[23] J. Edvardsson. A Survey on Automatic Test Data Generation. *2nd International Conference on Computer Science and Engineering, CSEN 1999*, pages 21–28, 2002.

[24] D. Willmor S. M. Embury. An Intensional Approach to the Specification of Test Cases for Database Applications. In *28th International Conference on Software Engineering, ICSE 2006*, pages 102–111, 2006.

[25] M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *2007 the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007*, pages 151–162, 2007.

[26] R. B. Findler and M. Felleisen. Contracts for Higher-Order Functions. In *Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2002*, pages 48–59, 2002.

[27] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering, FOSE 2017)*, pages 37–54, 2007.

[28] S. Fujiwara, K. Munakata, Y. Maeda, A. Katayama, and T. Uehara. Test Data Generation for Web Application Using a UML Class Diagram with OCL Constraints. *Innovations in Systems and Software Engineering*, 7:275–282, 12 2011.

[29] A. Kaehler G. Bradski. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, August 2009.

[30] A. Hori, S. Takada, H. Tanno, and M. Oinuma. An Oracle based on Image Comparison for Regression Testing of Web Applications. In *27th International Conference*

on Software Engineering and Knowledge Engineering, SEKE 2015, pages 639–645, 2015.

[31] A. Bergel J Ressia and O. Nierstrasz. Object-centric Debugging. In *34th International Conference on Software Engineering, ICSE 2012*, pages 485–495, 2012.

[32] C. Kaner, J Bach, and B. Pettichord. *Lessons Learned In Software Testing*. Wiley India Pvt. Limited, 2008.

[33] F. Kıraç, B. Aktemur, and H. Sözer. VISOR : A Fast Image Processing Pipeline with Scaling and Translation Invariance for Test Oracle Automation of Visual Output Systems. *Journal of Systems and Software*, 136, 2017.

[34] H. Kirinuki, H. Tanno, and K. Natsukawa. COLOR: Correct Locator Recommender for Broken Test Scripts using Various Clues in Web Application. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*, pages 310–320, 2019.

[35] T. Koju, S. Takada, and N. Doi. An Efficient and Generic Reversible Debugger Using the Virtual Machine Based Approach. In *1st ACM/USENIX International Conference on Virtual Execution Environments, VEE 2005*, pages 79–88, 2005.

[36] T. Kurabayashi, M. Iyama, H. Kirinuki, and H. Tanno. Automatically Generating Test Scripts for GUI Testing. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2018*, pages 146–150, 2018.

[37] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Robula+: an Algorithm for Generating Robust XPath Locators for Web Testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

[38] X. Li and M. Flatt. Medic: Metaprogramming and Trace-oriented Debugging. In *Workshop on Future Programming, FPW 2015*, pages 7–14, 2015.

[39] Y. Lin, J. F. Rojas, E. T. . Chu, and Y. Lai. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *IEEE Transactions on Software Engineering*, 40(10):957–970, 2014.

[40] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong. Feedback-Based Debugging. In *39th IEEE/ACM International Conference on Software Engineering, ICSE 2017*, pages 393–403, 2017.

[41] S. Mahajan and W. G. J. Halfond. Finding HTML Presentation Failures Using Image Comparison Techniques. In *29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014*, pages 91–96, 2014.

[42] S. Mahajan and W. G. J. Halfond. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015*, pages 1–10, 2015.

[43] K. Maruyama and M. Terada. Debugging with Reverse Watchpoint. In *Third International Conference on Quality Software, QSIC 2003*, pages 116–123, 2003.

[44] M. Monperrus. Automatic Software Repair: A Bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24, 2018.

[45] Martin Monperrus, Benjamin Danglot, Oscar Luis Vera-Perez, Zhongxing Yu, and Benoit Baudry. The Emerging Field of Test Amplification: A Survey. working paper or preprint, November 2017.

[46] K. Naitou, A. Tanikado, S. Matsumoto, Y. Higo, S. Kusumoto, H. Kirinuki, T. Kurabayashi, and H. Tanno. Toward Introducing Automated Program Repair Techniques to Industrial Software Development. In *26th Conference on Program Comprehension, ICPC 2018*, pages 332–335, 2018.

[47] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. Travassos. A survey on model-based testing approaches: a systematic review. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, WEASELech 2017*, pages 31–36, 2007.

[48] K. Pan, X. Wu, and T. Xie. Generating Program Inputs for Database Application Testing. In *26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011*, pages 73–82, 2011.

[49] K. Pan, X. Wu, and T. Xie. Program-input generation for testing database applications using existing database states. *Automated Software Engineering*, 22(4):439–473, December 2015.

[50] R. Ramler, T. Wetzlmaier, and R. Hoschek. GUI Scalability Issues of Windows Desktop Applications and How to Find Them. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 63–67, 2018.

[51] M. Rizun, J. C. Bach, and S. Ducasse. Code Transformation by Direct Transformation of ASTs. In *7th International Workshop on Smalltalk Technologies, IWST 2015*, number 7, pages 7:1–7:7, 2015.

[52] D. S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transaction on Software Engineering*, 21(1):19–31, 1995.

[53] S. Roy Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *2010 IEEE International Conference on Software Maintenance, ICSM 2010*, pages 1–10, 2010.

[54] P. Samuel and R. Mall. Slicing-based Test Case Generation from UML Activity Diagrams. *ACM SIGSOFT Software Engineering Notes*, 34(6):1–14, 2009.

[55] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In *13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005*, pages 263–272, 2005.

[56] IEEE Computer Society, P. Bourque, and R. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 2014.

[57] H. A. Souza, M. L. Chaim, and F. Kon. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR*, abs/1607.04347, 2016.

[58] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated Oracle Comparators for TestingWeb Application. In *18th IEEE International Symposium on Software Reliability, ISSRE 2017*, pages 117–126, 2007.

[59] J. Takahashi. An Automated Oracle for Verifying GUI Objects. *ACM SIGSOFT Software Engineering Notes*, 26(4):83–88, 2001.

[60] H. Tanno. Design and Implementation of Real-time Debugger for Game Programming (in Japanese). *Information Processing Society of Japan Transaction: Programming*, 1(2):42–56, 2008.

[61] H. Tanno and Y. Adachi. Support for Finding Presentation Failures by Using Computer Vision Techniques. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2018*, pages 356–363, 2018.

[62] H. Tanno, T. Hoshino, S. Koushik, and K. Takahashi. Test Data Generation for Integration Testing by Using Concolic Testing (in Japanese). (6):1–8, 2013.

[63] H. Tanno, T. Kurabayashi, X. Zhang, M. Iyama, Y. Adachi, S. Iwata, and H. Kirinuki. A Survey of Test Input Generation (in Japanese). *JSSST Computer Software*, 34(3):3_121–3_147, 2017.

[64] H. Tanno, M. Oinuma, and K. Natsukawa. Test Automation Technology to Promote Early and Frequent Releases of Software at Low Cost. 15, 2017.

[65] H. Tanno and X. Zhang M. Oinuma. Initial Database Generation for Integration Testing by Source Code Generation (in Japanese). (16):1–8, 2015.

[66] H. Tanno and X. Zhang. Automatic Test Data Generation Based on Domain Testing (in Japanese). *IPSJ SIG Technical Reports*, 2014(6):1–8, 2014.

[67] H. Tanno and X. Zhang. Test Script Generation Based on Design Documents for Web Application Testing. In *39th IEEE Annual Computer Software and Applications Conference, CPMPSAC 2015*, volume 3, pages 672–673, 2015.

[68] H. Tanno and X. Zhang. Test Script Generation Based on Software Design Documents (in Japanese). Technical Report 16, 2015.

[69] H. Tanno, X. Zhang, and T. Hoshino. Automatic Test Case Generation for Integration Testing (in Japanese). *IEICE Technical Report*, 110(227):37–42, 2010.

[70] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In *37th International Conference on Software Engineering, ICSE 2015*, volume 2, pages 717–720, 2015.

[71] H. Tanno, X. Zhang, K. Tabata, M. Oinuma, and K. Suguri. Test Automation Technology to Reduce Development Costs and Maintain Software Quality. 12, 2014.

[72] M. Terada. ETV: A Program Trace Player for Students. In *10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2005*, pages 118–122, 2005.

[73] T. A. Walsh, G. M. Kapfhammer, and P. McMinn. ReDeCheck: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages. In *26th International Symposium on Software Testing and Analysis, ISSTA 2017*, pages 360–363, 2017.

[74] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages. In *30th IEEE/ACM*

International Conference on Automated Software Engineering, ASE 2015, pages 709–714, 2015.

[75] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[76] T. Xie. Transferring Software Testing Tools to Practice. In *12th IEEE/ACM International Workshop on Automation of Software Testing, AST 2017*, pages 8–8, 2017.

[77] Y. Yanagisawa, K. Kourai, and S. Chiba. A Dynamic Aspect-oriented System for OS Kernels. In *5th International Conference on Generative Programming and Component Engineering, GPCE 2006*, pages 69–78, 2006.

[78] H. Yin, C. Bockisch, and M. Aksit. A Pointcut Language for Setting Advanced Breakpoints. In *12th Annual International Conference on Aspect-oriented Software Development, AOSD 2013*, pages 145–156, 2013.

[79] C. Zhang, D. Yan, J. Zhao, Y. Chen, and S. Yang. BPGen: an Automated Breakpoint Generator for Debugging. In *32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*, volume 2, pages 271–274, 2010.

[80] X. Zhang and T. Hoshino. Test Case Extraction and Test Data Generation from Design Models. In *5th ASQ World Congress for Software Quality, WCSQ 2011*, 2011.

[81] X. Zhang, H. Tanno, and T. Hoshino. Introducing Test Case Derivation Techniques into Traditional Software Development: Obstacles and Potentialities. In *Fourth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011*, pages 559–560, 2011.

[82] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study. In *32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 60–71, 2017.

Publication List

1. Haruto Tanno, Xiaojing Zhang and Takashi Hoshino.
   Design Model Based Generation of Initial Database for Testing.
   Journal of Information Processing.
   53(2):566–577, 2012.
   In Japanese.
   Corresponding to Chapter 3.

2. Haruto Tanno, Xiaojing Zhang, and Takashi Hoshino.
   Design-Model-Based Test Data Generation for Database Applications.
   In *23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2012*.
   pages 201–206, 2012.
   Corresponding to Chapter 3.

3. Haruto Tanno.
   Reduce The Number and The Size of Initial Database States for Testing Application.
   Journal of Information Processing.
   58(4):818–832, 2017.
   In Japanese.
   Corresponding to Chapter 4.

4. Haruto Tanno and Takashi Hoshino.
   Reducing the Number of Initial Database States for Integration Testing.
   In *37th IEEE Annual Computer Software and Applications Conference Workshops, COMPSACW 2013*.
   pages 59–64, 2013.
   Corresponding to Chapter 4.

5. Haruto Tanno, Yu Adachi, Yu Yoshimura, Katsuyuki Natsukawa, and Hideya

Iwasaki.
Region-based Detection of Essential Differences in Image-based Visual Regression Testing.
Journal of Information Processing.
28:268–278, 2020.
Corresponding to Chapter 5.

6. Haruto Tanno and Hideya Iwasak.
Suspend-less Debugging for Interactive and/or Realtime Programs.
In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019.*
pages 194–205, 2019.
Corresponding to Chapter 6.