Automated regression testing of database applications

Erik Rogstad

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo

October 2015

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1714

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Print production: John Grieg AS, Bergen.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the thesis defence. Kindly direct all inquiries regarding the thesis to the copyright holder or the unit which grants the doctorate.

Abstract

Ensuring the functional quality of database applications is a very important problem in software testing, yet few innovative solutions and empirical studies are reported on the subject. Database applications are widely adopted, for example in public administrations and banks, as they need to process large amounts of transactions efficiently for a large population and store large amounts of data. Such applications are often highly automated in order to efficiently cope with a large number of transactions, and are usually difficult to maintain and change. In order to preserve system quality through frequent system releases, a thorough, systematic, and automated regression test approach is needed for such applications, as they tend to provide core business value to their organizations.

The objective of this thesis is to help scale functional system-level regression testing of database applications through cost-effective automation. We propose a black-box approach that relies on classification trees to model the input domain of the system under test. We use the classification tree models as basis for automatically selecting abstract regression test cases, and either generate test data automatically according to the model specifications or rely on production data (data cloned from the production environment) that match the model specifications. Regression testing is carried out by running the selected test cases on consecutive versions of the system under test, while automatically capturing changes in the database state during system execution. The captured database manipulations for each test case are automatically compared across system versions, and test cases that deviate between two system versions are either due to anticipated changes in the release, or regression faults. The resulting deviations from a regression test are clustered according to their output characteristics, so that deviations resulting from the same change or fault (ideally) are contained in the same cluster. These clusters are then used to minimize the effort required to analyze deviations.

In order to evaluate our approach, we conducted a large-scale case study in a real development environment at the Norwegian Tax Department. The Norwegian Tax Department maintains several database applications, which are built on standard and widely used database technology and are representative of many such applications in public administrations. Together with the Norwegian Tax Department, we developed an industry-

strength tool in accordance with our proposed regression test approach. We applied the tool to the regression testing of their tax accounting system, thus evaluating its applicability and scalability on a large-scale database application with real changes and regression faults.

The results of our study showed that our proposed solution to regression testing helps mitigate risk when releasing new versions of a system, as it is more thoroughly, yet efficiently tested, causing less regression faults to be released. When our tool was applied for regression testing of eight consecutive releases of the subject system, it helped identify 60% additional faults to those found through regular testing. We regard this as a substantial contribution in terms of increased fault detection power. Furthermore, we made a thorough assessment of various strategies for selecting test cases based on classification tree models. When using existing production data sets as basis for regression testing, carefully selecting test cases according to their model partition coverage, can help reduce test effort dramatically. This is important in cases when running regression test cases are expensive, or when test results have to be manually inspected. For example, when selecting test cases according to our proposed selection approach, nearly 80% of the regression faults were captured when selecting only 5% of the test cases for execution. This is important in order to scale the regression test effort when the input domain, and thus the number of possible test cases are very large. To further add on the scalability of the regression test approach, our results show that clustering regression test deviations based on their output characteristics can help reduce the effort spent on analyzing such deviations. The clustering strategy turned out to be very accurate as it resulted in homogenous clusters (all deviations in each cluster match one change or fault) for all regression test campaigns assessed. This implies that testers can inspect one deviation only from each cluster and still remain confident of finding all regression faults. Moreover, we assessed the cost-effectiveness of various strategies for regression testing and found that combining combinatorial test suites with test suites conforming to the operational profile of the system under test was effective, as neither one alone is sufficient to find all kinds of regression faults.

In conclusion, we have proposed a novel and holistic solution to functional system-level regression testing of database applications, that automates many steps in the test process. The system under test is tested in a structured and systematic manner as we rely on model specifications to drive the test process. The regression test approach has been evaluated in a real and representative development environment and proved to be both effective in detecting regression faults and to scale for testing large database applications.

Preface

This thesis was undertaken at Simula Research Laboratory and the Department of Informatics, University of Oslo, with Lionel Briand as the principal supervisor. The work has primarily been carried out at the Norwegian Tax Department (NTD), who has been a sponsor and partner in the research project. Erik Arisholm was the principal supervisor during the first year, then Richard Torkar for the next half a year, before Lionel Briand, who had been subsidiary supervisor until then, became principal supervisor for the remainder of the PhD.

Stein Grimstad, Erik Arisholm and Trond Andreassen established the research project, partnering Simula Research Laboratory and NTD. Stein Grimstad was department head of the software engineering group of Simula at the time, Erik Arisholm was a senior researcher at Simula and Professor at the University of Oslo, and also in the startup-phase of establishing his own company within software testing, and Trond Andreassen worked as a senior engineer at NTD. Together, they shaped out the industrial research project, later known as the ATOS project, in which this thesis was undertaken.

Acknowledgement

I will do this chronologically. Sort of at least.

I will start with Erik Arisholm, who was principal supervisor during the first year of my PhD. He was the one inspiring me to take on a PhD. After I was interviewed and later on offered the position as a PhD-student in the project, I remember speaking to my wife about whether to quit my job and pursue this opportunity. I told her about the inspiring presence of that guy with all the wild-growing and slightly unkempt hair, who talked so encouragingly about the project. He gave me great faith in the project, and I told my wife I wanted to work with him, who was then set out to be my primary supervisor. Apart from my own motivation in advancing my academic background, his inspiring presence was a decisive factor in me deciding to do a PhD. And he delivered above expectations during his period of supervision. Practical, knowledgeable and launching bright ideas, while giving me the freedom to decide what to pursue. In short, we had a great time during that first year, working together in the ATOS project. He is undoubtedly one of the top, top guys in the software testing industry and I now enjoy working for him at Testify, where he has been incredibly generous and flexible in order for me to complete my thesis. Thank you, Erik!

Unfortunately, as Erik's business kicked off, he decided to move on from academia and quit PhD supervision. As much as that upset me, meeting the next principal supervisor in line, Richard Torkar, was another great source of inspiration. We first met over a beer in a pub, had a great talk, and staked out the course for the forthcoming period. Such great mood on that guy. And the enthusiasm he shows. Richard is highly skilled in conducting industry-driven research, hardworking, structured and very responsive and was there for me, whenever and wherever. Although his period in Norway remained relatively short, he brought a breath of fresh air into my PhD work. Thank you, Richard!

Again, unfortunately Richard decided to go back to Sweden for various reasons and could no longer continue as my supervisor. Despite the fact that Lionel had recently decided to move to Luxembourg and was gradually phasing out his engagements at Simula, he was very firm regarding my future PhD supervision at that stage: "From this moment on and until you graduate, I will be your principal supervisor. No matter what. Enough

changes." As being actively involved as subsidiary supervisor during the initial period, and then taking charge as principal supervisor for the remainder of the PhD, he has been the steady rock guiding me safely over the finish line.

Lionel. Given his merits, no one could have blamed him for being an arrogant prick. But he's not. Not even the tiny little bit. He is so devoted in his students work, that you wouldn't believe it. He cares. And contributes. He really does. And he is so passionate about research. A fellow PhD student of mine at Simula once injured his arm and couldn't write. Lionel then offered to sit down with him and type according to the oral dictation of the student, in order for him to progress his work. That's the kind of supervisor Lionel is. He takes active part in the research of his students. His capacity goes beyond most people and his knowledge about research within software testing is remarkable (and well documented), which makes him able to reflect and contribute on nearly any topic and give insightful feedback. While I entitled Erik as one of the top guys within the software testing industry, Lionel is "the one" within software testing research. Thank you, Lionel!

I feel ever so fortunate to have done a PhD in software testing under the supervision of Lionel Briand, Erik Arisholm and Richard Torkar. They all share the ability to inspire, which is one of the core qualities of a good supervisor. After having spoken to either one of them, I always feel wildly inspired to carry on. That's what you want from a supervisor. To all three of you, thank you so much for guiding me and inspiring me, and for all your pivotal contributions in my PhD work.

Apart from my supervisors, Ronny Dalberg at NTD has had great impact on my PhD. He is the one I have been working with on a day-to-day business while conducting industry-driven research. He has been instrumental in the practical application and implementation of research ideas at NTD. I have appreciated his company and he has been a great discussion partner throughout. Thank you, Ronny! I would also like to thank the administrators of the ATOS project at NTD, Hilde Lyngstad, Marianne Amundøy Vikdal and especially Marianne Rynning for good and flexible management of the project.

Special thanks to Simula Research Laboratory, Simula School of Research and Innovation and my colleagues there for providing an excellent work place throughout my PhD.

I would also like to thank my family, and in particular my wife, Ida, for her loving support throughout the PhD. Especially during the latter stages, when she has taken care of the kids on many occasions, while I had to put in extra shifts. You have been incredible! And thanks to my two lovely daughters, Julie and Emilie, who were both born during the PhD period, and have brought so much joy and happiness into my life.

Erik Rogstad, September 2015

List of Publications

The following papers are included in this thesis.

1. Industrial Experiences with Automated Regression Testing of a Legacy Database Application
E. Rogstad, E. Arisholm, L. Briand, R. Dalberg, and M. Rynning
Published in the proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), pp. 362-371, 2011

2. Test Case Selection for Black-Box Regression Testing of Database Applications
E. Rogstad, L. Briand, and R. Torkar
Published in Information and Software Technology (Elsevier), volume 55, issue 10, pp. 1781-1795, October 2013

3. Clustering Deviations for Black Box Regression Testing of Database Applications
E. Rogstad and L. Briand
Accepted for publication in IEEE Transactions on Reliability, 2015

4. Cost-effective Strategies for the Regression Testing of Database Applications: Case study and Lessons Learned
E. Rogstad and L. Briand
Accepted for publication in Journal of Systems and Software (Elsevier), 2015

The above papers are self-contained and therefore some information might be repeated among them. Some acronyms and terminology may also differ across papers.

My contributions
I was the lead author on all papers, and thereby main responsible for conducting the research and writing the papers. My supervisors contributed in all phases of the work, in particular in the study planning and during paper writing, whereas I executed the studies. Ronny Dalberg at the Norwegian Tax Department (industry partner) helped transforming research ideas into applicable tool implementations.

Contents

List of Figures

List of Tables

Part 1: Summary of the Thesis

Chapter 1

Introduction

The objective of regression testing is to improve confidence that changes behave as intended and that they have not adversely affected unchanged parts of the software [19]. Regression testing plays an integral role in maintaining the quality of subsequent releases of software, but it is also expensive, accounting for a large proportion of the costs of software production and maintenance [2, 27, 23]. Because regression testing is important, but expensive, the topic has been much researched in order to achieve increased effectiveness and efficiency. In particular a great deal of work has been performed on devising and evaluating techniques for selecting, minimizing, and prioritizing regression test cases [44]. However, as noted in a survey on regression testing by Harrold and Orso [19], technology transfer from research to the practice of regression testing is very limited. Although there are a number of papers on evaluation of regression testing techniques using controlled experiments (e.g., [3, 13, 16, 31, 32, 37, 38]), only a few of these empirical studies (e.g. [29]) have been performed on real-world, large-scale systems or have shown benefits in practice. To reduce the gap between research and industry within regression testing, there is a need for additional empirical studies in real development environments to address practical regression test problems and evaluate their solutions in an industrial context.

Yoo and Harman [44] conducted an extensive survey on regression testing minimization, selection and prioritization, capturing the main research results around regression testing. The survey shows that the vast majority of research focuses on white-box testing strategies, which are primarily targeted towards achieving structural and change coverage of an application. When the objective is to test system level functionality, testers tend to prefer black-box approaches, based on the system specification. Also, in many situations, white-box testing is not practical due to lack of proper tool support or their lack of scalability to large systems. In other cases, it is not even applicable if there is no direct access to the source code or third party components. Furthermore, if testers have limited technical expertise regarding the system implementation, they may prefer to

verify system functionality based on the specifications rather than the source code. Under such circumstances a black-box approach to regression testing must be adopted, which is a challenge given the lack of such techniques proposed in the research literature.

More specifically, the functional regression testing of database applications has been given even less attention. Yet, there exist many large database application systems with a long, often unforeseeable life span, as they continue to provide core business value to their organizations. Many such applications are built on old technology and are constructed without particular considerations for testability. Their functional quality is typically assured through extensive manual testing during construction, but as the size and complexity of the system grow, the regression test effort exceeds what can be effectively handled manually. Thus, finding practical ways of introducing test automation to better scale regression testing for large database applications is important to sustain the stability of core business systems.

In practice, it is very hard to build a full-fledged automated regression test solution for system level testing. For example, for database applications with many highly complex queries, it is difficult to build precise test oracles. Thus, we capture a set of test case executions of the system under test, under the assumption that it currently works correctly, and then use replay runs after modifications to identify deviations and thus potential regression faults. With such an approach, we use the baseline execution as an automatically generated test oracle. This is effective in terms of automatically separating test cases with unchanged results from those deviating from the baseline results, which is important to ensure effective regression testing. However, the manual inspection of deviations from the baseline is still necessary in order to separate deviations that result from correct changes from those that are due to regression faults. Finding ways to cope with the many deviations that can be observed when running regression tests on a new version of a system is a highly important problem in practice. Thus, further measures are needed in order to ensure scalability of regression tests and help testers when the number of deviations exceeds their inspection capacity.

One such measure is to reduce the size of the regression test suite by selecting a subset of test cases in such a way as to maximize the likelihood of detecting regression faults. This is a core problem in regression testing, which is particularly acute when the test results must be manually checked or when running test cases is expensive. The test case selection should aim at reducing the test suite size and consequently the test effort, while maximizing the number of distinct regression faults in the test output.

Even with efficient test case selection, the number of regression test deviations may still be large, depending on the scope of the system changes. However, many deviations are likely to cover duplicate regression faults or be the result of the same changes. Another

measure to ensure scalability is then to group regression test deviations, such that each group of deviations ideally cover one regression fault or change. A precise mechanism for doing so would enable testers to focus their efforts towards covering groups of deviations rather than individual deviations and potentially gain significant savings in inspection effort.

Yet another challenge is to provide test data that satisfies the test specifications in order to execute actual regression tests. A common approach when testing database applications, at least for mature systems, is to rely on production data. In terms of test quality there is no better option than using production data, but they may be rigid to work with, they provide unpredictable coverage that may not fit the test requirements at hand, and confidentiality issues have to be sorted. For test automation purposes it is often more desirable to generate test data, as it tends to offer more flexibility regarding what to test at a given point in time and also provides more predictable test coverage. However, establishing procedures for generating such test data is associated with a high initial cost. Whether to opt for production data or generated data is not clear-cut, but either way mechanisms are needed in order to select or generate data according to a given test specification.

So to ensure scalability of regression testing and to keep the manual effort at a minimum, it is important to have a structured specification of test cases, automated procedures for carefully selecting or prioritizing test cases, automated support for collecting a test baseline and compare test results across system versions, automated ways of providing test data needed to execute the regression tests, and automated support for regression test deviation analysis.

Contributions

The work in this thesis was motivated by test challenges at The Norwegian Tax Department and has focused on test automation in the context of functional black-box regression testing of database applications, based on model specifications from classification tree models. The main contributions of the thesis are:

1. A novel and practical approach to automated, system-level regression testing of database applications, that uses classification tree models for black-box test specifications.

2. An industrial case study of the proposed regression test approach, by applying a novel tool implementation, DART ([DA]tabase [R]egression [T]esting), to the business critical batch jobs of a large database application in a public-service setting.

3. A thorough investigation of strategies for selecting test cases generated from classification tree models, evaluated in the context of black-box regression testing of database applications.

4. The definition and evaluation of a clustering strategy for grouping regression test deviations according to their root causes in order to help scale their inspection and analysis.

5. A practical and novel approach for matching production data against classification tree models in order to (1) detect model coverage, and (2) reduce the level of redundancy, and thus test effort, by selecting a subset of test cases for test execution.

6. A practical approach for automatic test data generation based on classification tree models.

7. An assessment of the combination of production data and generated test data for regression testing, following various combinatorial test strategies based on classification tree models.

In the list above, by "practical" we mean scalable to large systems, while relying on information that can realistically be provided by test engineers or domain experts.

Structure of the Thesis

This thesis is a collection of papers and the remainder of the thesis is organized in two parts.

Summary (Part I): Chapter 2 gives an introduction to relevant background information needed to understand the thesis. Chapter 3 provides an overview of the proposed regression test methodology, by tying together the various solutions from the individual papers. Chapter 4 presents the research methodology and Chapter 5 summarizes the key results of the thesis. Chapter 6 outlines future research directions, before providing concluding remarks in Chapter 7.

Papers (Part II): This part includes the four papers of the thesis, either submitted or accepted for publication in international journals and peer-reviewed conferences. Paper 1 presents the regression test approach and evaluates its fault detection capabilities, i.e. contribution 1 and 2 from above. Paper 2 covers contribution 3 and investigates various strategies for selecting test cases. Paper 3 addresses the problem of coping with the many regression test deviations resulting from regression testing (contribution 4). Paper

4 presents practical approaches for selecting or generating test data needed to execute regression tests and assesses their relative cost and effectiveness, thus covering contributions 5 to 7.

Chapter 2

Background

In this section, we provide background information on the main concepts involved in this thesis, including database applications - the type of software systems our work focuses on, regression testing in the context of database applications, classification tree modeling - used for black-box specification of the input domain of the system under test, and test case selection techniques.

2.1 Database applications

A database application is a computer program whose primary purpose is entering and retrieving information from a computerized database [40]. Early examples of database applications were accounting systems and airline reservations systems, and although database applications are now widespread across most application areas, some of the most complex database applications remain accounting systems, such as SAP [22, 6]. Database applications are also commonly used in banking and by public administrations, who typically need to process large numbers of transactions for a large population and store large amounts of data.

Modern database applications usually have a web interface, which is used by end users to interact with the system. Additionally, a common characteristic of such large database applications is the ability to process large amounts of data effectively, often through batch processing. Batch processing is the execution of a series of programs ("jobs") without manual intervention [39]. Such batch processes are efficient as the program can run once for many transactions reducing system overhead. They avoid idling computer resources with manual interventions and the batch processes can be scheduled to time periods where computer resources are less busy.

Oracle is the world market leader both in terms of application platforms and database management systems [15]. In a traditional Oracle database application, the data is stored

in a relational database, the business logic is written using SQL and PL/SQL programming, with Oracle Forms user interfaces. In recent years, after Oracle acquired Sun Microsystems, it is getting gradually more common to have Java applications running on top of an Oracle database, rather than using stored procedures and having a large number PL/SQL packages in the database.

2.2 Database regression testing

Regression testing is the activity of testing software after it has been modified to gain confidence that the newly introduced changes do not obstruct the behavior of the existing, unchanged parts of the software [44]. There are a number of challenges related to regression testing, such as identification of obsolete test cases, regression test selection, prioritization and minimization and test suite augmentation [19]. Yoo and Harman [44] conducted a survey on regression testing minimization, selection and prioritization, constituting nearly 200 papers. It encompasses the main research results around regression testing, addressing the problems of identifying obsolete, reusable and re-testable test cases (selection), eliminating redundant test cases (minimization) and ordering test cases to maximize early fault detection (prioritization). The survey shows that the bulk of existing work focuses on white-box testing strategies, concerning relatively small stand-alone programs written in C or Java, or for spreadsheets, GUIs and web applications. The techniques surveyed assume an already existing, effective test suite on which to select, minimize and prioritize test cases for the regression test. In practice, this is not always the case and more fundamental steps are required, namely how to collect a test baseline, and how to perform regression testing.

In the context of database testing, Chays et al. [8] noted the lack of uniform methods and testing tools for verifying the correct behavior of database applications, despite their crucial role in the operation of nearly all modern organizations. Most of the literature in the field is aimed at assessing performance of database management systems rather than testing the database application system for functional correctness, let alone regression testing. The authors proposed a framework for functional testing of database applications called AGENDA [7, 9, 12]. However, the framework was not intended for regression testing and the ideas have only been evaluated on smaller examples and seem unlikely to scale to fit industrial needs.

The most relevant work found on the topic of regression testing for database applications was the SIKOSA project [17, 4, 18]. The authors proposed a capture-and-replay tool for carrying out black-box regression testing of database applications, which is a sensible approach because it is hard to build a precise test oracle for database applications

with very complex queries. The work was restricted to checking input-output relations of database applications, as they stated that checking the state of the database after each test run was prohibitively expensive and difficult to implement for black-box regression testing. The SIKOSA project provided some experimental performance measures for their tool, but did not refer to any evaluations regarding fault-detection effectiveness or cost-effectiveness, let alone in an industrial setting.

2.3 Classification tree modeling

Classification tree modeling is a category-partition-based modeling technique [30] that is typically used for modeling configuration parameters [10, 41, 43], or the input domain of the system under test, i.e. input parameters [34] or properties of the system under test [11]. A classification tree model is a tree structure where all relevant distinguishing properties of the system under test are captured at the desired granularity level. For example, properties related to the input domain that may affect the behavior of the system under test can be identified, and split into equivalence classes following usual black-box testing strategies such as boundary value analysis. An artificial example of a classification tree model, made using the tool CTE-XL [26], is given in Figure 2.1. The model is visible in the upper right corner, where relevant properties of the input domain are modeled as *classifications* (e.g. *Property B: Nr of X*), and split into *equivalence classes* (e.g. the ranges *1-4* and *5-10*). As shown, it is also possible to conform to a hierarchical tree structure, by modeling sub-properties under an equivalence class.

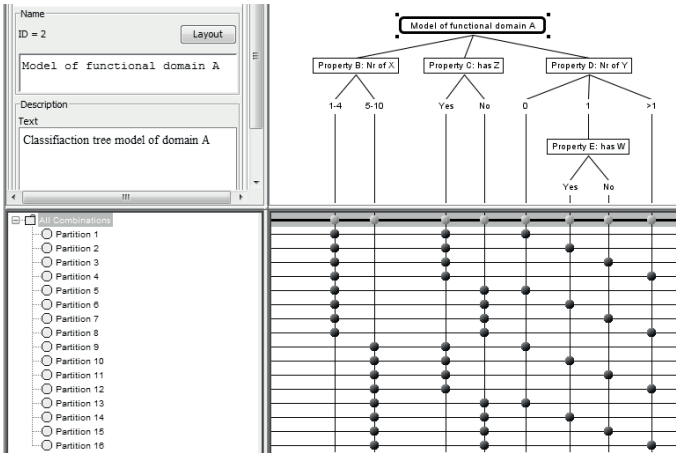


Figure 2.1: An example of a classification tree model in CTE-XL and the generated partitions (combinations of equivalence classes) that form abstract test cases.

Classification tree models are used as basis for combinatorial testing, which aims at systematically testing valid combinations of equivalence classes according to a specified degree of coverage. Pair-wise and three-wise are examples of common coverage criteria in combinatorial testing, which are described in more detail in Section 2.4.2. Given a classification tree model and a coverage criterion, a set of model partitions can be generated, as shown in the lower part of Figure 2.1. For the simple model presented here, we have generated all possible combinations, and all resulting partitions are gathered in a set labeled "All Combinations". A partition is a specific combination of equivalence classes, which is visible in the figure as a line with a mark (dot) for each equivalence class covered by the partition. In our context, a partition corresponds to an abstract test case, or the specification of a test case, and a collection of partitions corresponds to a test suite specification.

2.4 Test case selection

Test case selection is the activity of selecting a subset of test cases for execution, in such a way that the fault detection capabilities of the subset are maximized. We present below two approaches to test case selection that are relevant for the thesis, namely *similarity-based selection* and *combinatorial testing*.

2.4.1 Similarity-based selection

Similarity-based test case selection attempts to select test cases that are as diverse as possible, based on similarity measures. A similarity-based selection strategy consists of three parts:

- Encoding - how to encode your test cases for comparison
- Similarity function - how to measure similarity between test cases
- Selection algorithm - how to select test cases on the basis of their similarity

In other words, the concept of similarity-based selection is about finding a way of representing test cases in such a way that their similarity can be measured, and then use approximation algorithms to select test cases as diverse as possible, under the hypothesis that increased diversity among tests leads to increased fault detection rates. Similarity-based selection has shown promising results, both in the context of model-based testing [21] and labeled transaction systems [14].

2.4.2 Combinatorial testing

Combinatorial testing is about systematically testing combinations of parameter equivalence classes according to a specified degree of coverage. The aim is to reduce the number of test cases to execute, while retaining a broad and systematic coverage that maximize the chances of revealing faults with reduced effort. Combinatorial testing relies on the assumption that most faults are triggered by either a single parameter value or by a combination of a few parameters' values [5]. For example, Kuhn et al. studied the faults in several software projects, and found that all the known faults are caused by interactions among six or fewer parameters [24, 25]. In such cases, combinatorial testing can be very effective, with performance approaching that of exhaustive testing while significantly reducing the number of test cases.

Combinatorial testing is based on a model of the system under test that typically contains the parameters that may affect the system under test, the values or equivalence classes that that are defined for each parameter, and constraints between parameter values. The constraints can be used to exclude combinations that are not meaningful from the domain semantics. Classification trees, as described in Section 2.3, are examples of such models. These models are then used as basis for generating a test suite specification by selecting test cases that combine values of different parameters according to a combinatorial test strategy. The most common combinatorial test strategies are pair-wise and three-wise, which can be defined as follows:

- The *pair-wise* generation criterion is satisfied if every possible pair of equivalence classes is covered by at least one test case specification in the resulting test suite. Following the example from Figure 2.1, the equivalence classes *1-4* and *Yes* of *Property B* and *Property C*, respectively, should appear at least once in the test suite specification [28, 42, 20].

- The *three-wise* generation criterion is satisfied if every possible triple of equivalence classes is covered by at least one test case specification in the resulting test suite. Following the example from Figure 2.1, each of the equivalence classes *1-4*, *Yes*, and *>1* of *Property B*, *Property C* and *Property D*, respectively, should appear at least once in the test suite specification. In the example case, there would be no difference between *three-wise* and *all combinations*, because the model only contains three model properties at the top level [24, 33].

Combinatorial testing can reduce the number of tests radically, while still ensuring that every *N-wise* combination of equivalence classes is covered. As an example, consider a program with 10 parameters taking two values each. The total number of test cases to

test exhaustively is then $2^{10} = 1024$. An optimal algorithm for three-wise combinatorial testing would result in 13 test cases.

As being a specification-based testing technique, combinatorial testing requires no knowledge about the implementation of the system under test. Also, the specification is "lightweight" since it only requires knowledge about the basic system configuration in order to identify the input parameters and their possible values.

Chapter 3

Automated Regression Testing of Large Database Applications

The goal of this thesis is to handle regression test challenges that are faced, in practice, when evolving large database applications during development and maintenance. In this section, we will first explain the challenges, and then describe our proposed solutions to solve the problem.

3.1 Regression test challenges

As mentioned in Section 2 a common characteristic of large database applications is the ability to process large amounts of data effectively, often through batch processing. While batch processes enable efficient processing of large amounts of complex transactions, performance has traditionally been a stronger non-functional driver than testability during their construction. Thus, testing them is challenging. For example, batch processes are hard to control and observe during testing. They can be started and then run to completion without any further mechanisms of control. That means the input of the batch process can be controlled, and the end result checked, but what happens in between is difficult to observe, and even more difficult to control. Moreover, the input domain consists of both system inputs and the state of the database, thus accounting for a high degree of input variation leading to a wide range of test scenarios. As the batch programs are completely automated processes, they also tend to be complex, consisting of a large number of tightly integrated sets of operations. All of these factors make the regression testing of database applications, and in particular batch processes, very challenging.

The Norwegian Tax Department maintains several database applications, including SOFIE, the tax accounting system of Norway. SOFIE is an example of a large and complex database applications that handles tax calculations and tax transactions for all

taxpayers in Norway, while also keeping track of large amounts of data, including historical tax data. The Norwegian Tax Department realized the test challenges mentioned above, and concluded that their current manual testing practice were not deemed adequate for the effective regression testing of their tax accounting system. Their database applications provide core business value to their organization, and they had managed to provide reasonably dependable system features through extensive internal testing and a large user base over a long period of time. Yet, the systems are complex and hard to test, making them error prone following the many changes forced upon by changes in taxation laws and regular maintenance. This motivated NTD to establish a cooperation project with Simula Research Laboratory, in order to come up with more efficient ways to perform regression testing of database applications by introducing a higher degree of test automation. This initiative set the frame and formulated the problem for which this thesis was undertaken.

3.2 Overview of regression test solution

Our response to the regression test challenges mentioned above was to develop a practical approach and tool (DART) for functional, black-box regression testing of database applications. The main reason for choosing a black-box approach to regression testing was the fact that the testers in the project had limited technical expertise regarding the system implementation, and preferred to verify system functionality based on the specifications rather than the source code. Hence, we adopted a black-box approach to regression testing that does not require source code analysis. However, black-box approaches also have the advantage of being more applicable when there is a lack of proper tool support for source code analysis in a given technology context, or when there is no direct access to source code or third party components.

Figure 3.1 shows an overview of the approach to regression testing of database applications. The approach is twofold, namely (1) the specification and selection of test cases along with the setup of test data needed to execute the regression tests (left part of the figure) and (2) the execution of the regression tests (right part of the figure). The details of the approach are described in subsequent sections below but overall, it is a capture-and-replay approach, similar to what has been more commonly used for GUI testing, to automatically identify differences (referred to as regression test deviations) between the results of two identical test runs (same input and initial state) on different versions of the system under test. Because it is hard to build a precise test oracle for database applications with very complex queries, a more practical strategy is to capture a set of test case executions of the system under test, under the assumption that it currently works

correctly (the baseline), and then use the replay run after modifications (the delta) to identify deviations and thus potential regression faults.

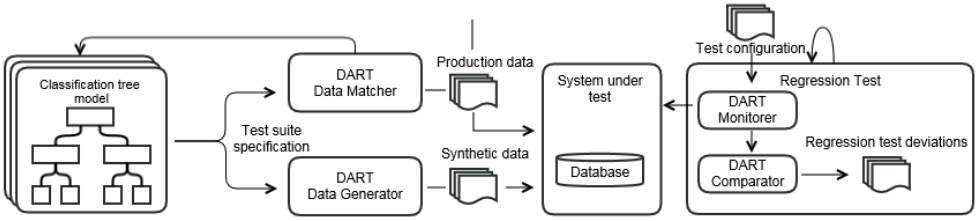


Figure 3.1: Overview of regression test approach.

We rely on classification tree models to model the input domain of the system under test (SUT), in order to obtain a practical and scalable solution. The test models enable us to systematically approach the variation in the input domain, as we model properties of the input domain and use test case generation algorithms to ensure predictable and complete model coverage. The abstract test specifications generated from the test models are used to drive the generation of test data and select production data for test execution.

Although our regression test approach can be used to test any type of database application, it is particularly suited for testing batch processes (or similar types of database intensive programs) that process large amounts of data automatically, thus making manual testing impractical.

3.2.1 Test specification and test case selection

In order to remain systematic when testing, we needed a specification-based, black-box testing technique to help specify test input data (test cases), based on an analysis of the input domain of the system under test, e.g. a batch process. There are many suitable tools for this purpose, but we found that the classification tree modeling technique and the supporting tool CTE-XL [26], was both easy to use and scaled up to the kinds of input domains under consideration (e.g., more than 100 categories or classifications in one model). As mentioned in Section 2.3, classification tree modeling is built on the well-known category-partition approach [30] and is a common approach to combinatorial test design. The input domain (i.e. input parameters or properties of the system under test) is modeled as a classification tree, which in turn is used to generate a combinatorial test suite specification, and thus aligned well with our needs.

To be able to use the models for anything practical, we needed to integrate them with our regression test tool, DART. DART has its own database, where everything related to the regression tests are stored, i.e. test configurations, test executions and test results, as described in Section 3.2.2. We extended the database schema to also

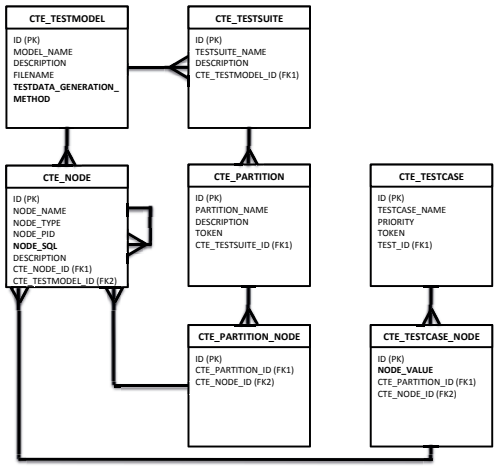


Figure 3.2: ER-diagram for CTE-XL models in DART.

include tables equipped to store classification tree models. CTE-XL stores the models as XML-files, which we parse and store into the DART database complying with the entity-relationship model shown in Figure 3.2. A test model (`cte_testmodel`) contains a set of nodes (`cte_node`), i.e. the classifications and equivalence classes in the model, and one or more test suite specifications (`cte_testsuite`), i.e. sets of partitions. Each test suite specification contains several partitions (`cte_partition`), each containing a set of nodes (`cte_partition_node`), that is the specific equivalence classes covered by the partition (the dots on each partition line in Figure 2.1). `cte_testcase` and `cte_testcase_node` map to `cte_partition` and `cte_partition_node`, and capture actual test cases in the system under test, as opposed to abstract test cases in the test model. Having the CTE-XL-models integrated into our test tool enables us to use the models for test automation purposes.

We use these test models to (1) systematically drive test data generation (DART Data Generator in Figure 3.1) and (2) to match production data against the model in order to determine coverage (DART Data Matcher in Figure 3.1), either a) to select production data for testing or (b) to analyze the operational profile of a particular functional area of the system under test.

Selecting test data based on test models

When using production data as basis for testing, the selected test data will vary among (regression) test campaigns. In order to remain systematic when testing, we use the test models to drive the selection of test data based on the coverage of partitions. By matching the test data against the classification tree models, we are able to (1) detect which model

partitions and model properties are covered (and not covered), and (2) reduce the level of redundancy, and thus the test effort, by selecting a subset of test cases for test execution. In short, our selection strategy selects, in a balanced way, test cases from all covered partitions in the classification tree model, while attempting to select the most diverse test cases from each partition. The full details of the test case selection approach is presented in the second paper (Chapter 2 in Part 2)

The classification tree models tend to be relatively high-level representations of the input domain of the system under test. Therefore, a gap exists between the abstract test cases defined by the model and executable test cases. In other words, there is no one-to-one relationship between the properties captured in the model and the concrete database fields in the database. The modeling is driven from a functional point of view to capture the variability of the input domain of the system under test, and is not concerned with the particular details of the system database. However, the gap between the model and the database has to be addressed somehow, to ensure that we can match actual test data with the model, and furthermore to generate executable test cases from the abstract test cases.

When it comes to matching test data against the model, we have chosen to solve the mapping by extending the definition of the classification nodes in the model with an SQL query that maps the model property with its concrete value(s) in the database. The attribute `node_sql` on the `cte_node` entity, which is highlighted in Figure 3.2, holds this additional mapping information. The SQL query is built up in such a way as to extract information from the database regarding the value of the model property for a given test case, or set of test cases.

Following the example from Figure 2.1, each of the four classification nodes (Property B, C, D, and E) in the model, would have an attached SQL mapping query. For example, *Property B: Nr of X* would have an SQL mapping query that returns the value of *Nr of X* for an actual test case, and maps this value to the test case. The mapping between a test case and its model property values is given via the `cte_testcase` and `cte_testcase_node` entities. The set of test cases that should be examined is held by the `cte_testcase` entity. For each leaf node in the model (equivalence classes at the bottom level of the tree) corresponding to the actual value of a property, a link is established between the test case and the model node via `cte_testcase_node`. For example, if a test case has the value 3 for *Property B: Nr of X*, then a `cte_testcase_node` is created for the test case, with a reference to the leaf node with the value range 1-4, and the actual value of the test case (3) is stored in the attribute `node_value`.

Based on this strategy for matching test data with a test model, we can implement a generic solution, independent of the specifics of each individual test model. The mapping

SQL query can be seen as an extension to the modeling effort, but once established, the matching of test data with the test models is general. After having conducted the matching, we can bring the results back to CTE-XL (by extending the XML file of the model), in order to get a visual representation of coverage obtained by the test suite.

The capability of matching data against a classification tree model enables us to do the following:

- Select production data for regression testing: Given a set of data available for a regression test, we match the data against the model to determine the model partition coverage. Typically, many test data instances match the same partition, and by selecting tests in a balanced way across model partitions, we are able to reduce redundancy, and thus test effort, while maximizing model partition coverage. The results from Paper 4 (Chapter 4 in Part 2) show that the expected level of redundancy for tests based on production data is high, while the results from Paper 2 (Chapter 2 in Part 2) shows the increased efficiency of the regression tests when selecting tests carefully according to partition coverage.

- Analyze operational profiles of the system under test: By matching production data for a particular functional area of the system under test, we can learn the distribution of data across model partitions and equivalence classes. For example, we can determine that 30% of the data in the operation of the system fall under a certain partition, while the value of a certain property has a 90/10 per cent distribution across its two equivalence classes. In turn, such an analysis could be used as input to testing, in order to perform operational profile testing.

Generating test data

Classification tree modeling offers a systematic and well-defined frame for generating test data. It provides a clear overview of the input domain of the system under test, from which to generate test case specifications that ensure combinatorial model coverage of a specified degree (e.g. pair-wise, three-wise). Thus, we have also used the classification tree models to drive the generation of test data. Using the same classification tree models, both for analyzing coverage of production data used for testing and for the generation of test data, enables us to consider a hybrid solution. For example, a test campaign can primarily be based on production data, while complementing it with generated test data when test specifications cannot be matched with production data.

However, generating executable test cases from a test case specification is a far more complex affair than to extract the actual values of existing data. Even a "simple" test case will most likely require a large extent of data populated in large parts of the relational

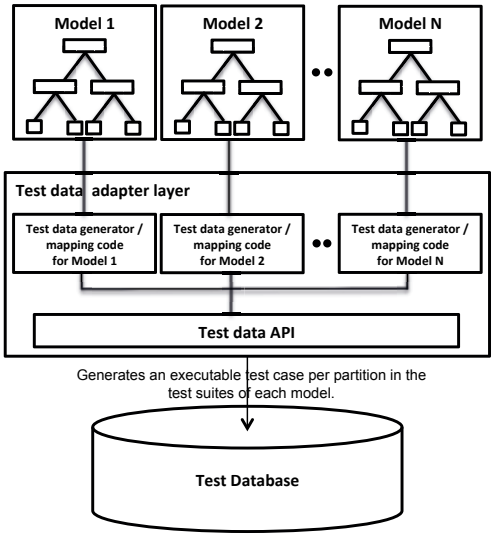


Figure 3.3: Generated test data using an adapter layer, that interprets a model and uses a test data API to creates executable test cases in the test database.

database. Thus, we have chosen to solve the mapping between the abstract test cases defined in the model and the executable test cases stored in the database, by implementing an adapter layer for test data generation, following the architecture in Figure 3.3. The adapter layer consists of a general test data API and one test data generator for each test model. The test data API holds general functionality for populating the various tables in the relational database with data, whereas each test data generator interprets a test model and its partitions and calls on the test data API to populate each test case with the properties and values specified by the model partitions. Additionally, we have defined a set of variables, both at the level of a test suite specification and a partition, which the tester can override in order to customize the test cases. These variables represent details about the test cases, that are not regarded as important for the characterization of the input domain, and thus are not captured in the model, but nevertheless capture values a tester might want to tweak in a test. Examples of such values can be the taxpayers municipality, or the year of the tax calculation, which a tester will typically change over time.

The tester can choose to generate test cases for an entire test suite specification, or for an individual selection of partitions, and the result of the generation is a set of test cases stored in the test database, ready for execution. An important consideration regarding the test data generated is that they are independent of the state of the database. In practice this is obtained by generating the test data in a completely synthetic manner, with no

relation to other data in the database. Then the test case is expected to behave the exact same way, when executed on the same program versions, independent of the evolution of the data in the test database. This is important in order to ensure comparable test executions across system versions.

Being able to generate test data for any given test case specification from a classification tree model gives the tester an enormous flexibility in terms of executing regression tests. For every regression test, a new equivalent set of test data is generated based on a given test specification, which makes the test executions comparable across system versions. Tests on synthetic test data are less rigid than tests on production data and thus more suited for test automation purposes. Nevertheless, as shown in Paper 4, generated test data does not fully replace tests based on real production data, thus both are needed in order to ensure as complete regression tests as possible.

3.2.2 Regression test

We have proposed a partly automated regression test procedure and tool (DART) tailored to database applications. The basic principle of the tool is straightforward: It compares executions of a changed version of the program against the original version of the program and identifies deviations, that is differences in the way the database is manipulated between the two executions. In each test execution, the database manipulations are logged according to a specification by the tester indicating the tables and columns to monitor. The database manipulations from each execution are compared across system versions to produce a set of deviations, which indicate either correct changes or a regression faults.

The strength of this approach is that it provides the ability to verify the entire set of test data executed by a database application automatically. As an example, let us say we execute a batch running the tax calculation for 10.000 taxpayers, each constituting a test case. Manually verifying 10.000 tests is far beyond what a tester can realistically handle. Therefore, one would have to pick out a small sample to analyze based on qualified guesses whereas the rest of the 10.000 tax calculations would remain unattended and pose substantial risk to the system release. However, with the regression test procedure suggested above for database applications, all the 10.000 tax calculations will automatically be compared against a previous execution to separate the test cases that deviated from the ones that did not.

Note that DART can be used to identify regression faults in any system or program unit performing Create, Read, Update and Delete (CRUD) operations on a database, and is not restricted to batch testing only. But in our industrial research context the system under test has consisted of batch processes that perform complex CRUD operations on a database, guided by business logic that implements sequences of taxation laws and rules.

Figure 3.4 shows the main steps in the testing process with DART. In the following sections, these steps will be described in detail.

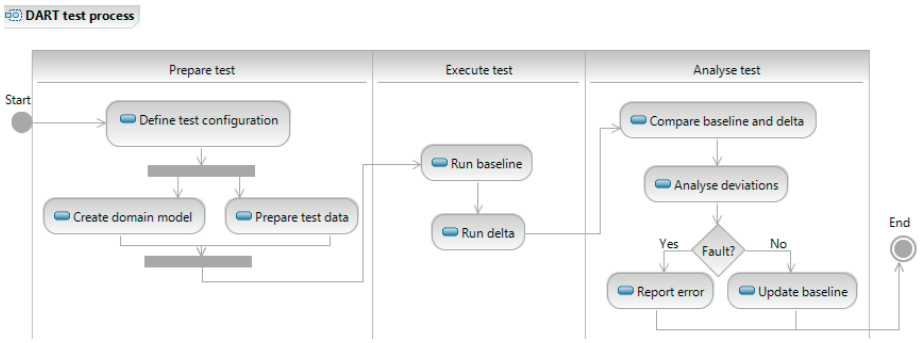


Figure 3.4: A UML Activity Diagram of the DART regression test process.

Running example

Throughout the description of DART, a running example will be used to demonstrate the various steps of the test process. The example is intentionally kept very simple but nevertheless illustrative. The system under test used as example is the program P shown in Figure 3.5. We use a Java-like syntax augmented with directly executable SQL statements in order to facilitate the understanding of readers not acquainted with PL/SQL. It is a program that contains features for maintaining customer orders, more specifically adding and deleting items from a customer order. As an example execution of the program, one item is added to a customer order, while an item is removed from another customer order in the main method.

The relational entity model of the example program is shown in Figure 3.6, along with the initial state of the database prior to test execution. It consists of three entities containing information about customers and their orders. A customer can have zero to many orders with zero to many items.

Test configuration

A test with DART is set up by selecting the database tables and more specifically the table columns to monitor during the test execution. DART obtains and presents the database schema(s) of the system under test and a test engineer selects the ones to monitor during the test execution. In our example the tester would be presented with the three tables Customer, Order and Item, which all are a part of the database schema for program P. Since the program P performs operations on the two tables Order and Item, these are the

Program P

```
void removeItemFromOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                 where Order.orderNr = orderNr;
  if(orderId != null) {
    delete from Item where Item.itemName = itemName and
    Item.orderId = orderId;
    update Order set Order.changedDate = tomorrow
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void addItemToOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                 where Order.orderNr = orderNr;
  if(orderId != null)
    insert into Item (ItemName, OrderId) values ( itemName
orderId);
    update Order set Order.changedDate = tomorrow
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void main() {
  addItemToOrder(12345, "USB stick");
  removeItemFromOrder(34567, "Mouse");
}
```

Figure 3.5: The example program P.

ones that make sense to monitor while testing P. The tester selects the two tables and more specifically the underlying table columns to monitor.

Additionally the test engineer specifies how CRUD-operations on the selected entities should be grouped together as "logical test cases" based on a meaningful, common test case identifier, e.g., a social security number. The identifier is defined using table attributes such as primary keys, foreign keys and/or SQL queries. The goal is to logically group related rows in the tables and monitored in a test execution to facilitate the comparison between the baseline and delta test executions. A meaningful common test case identifier in our example would be the customer name (assumed to be unique), as all orders and items can be traced back to its customer. In that case one customer will make out one test case and all data manipulations that are logged during test execution will be grouped by customer name. A test configuration for program P would then look like the one shown in Table 3.1.

In summary a test configuration denotes the set of table columns to monitor during test execution and the corresponding specification of the test case grouping scheme.

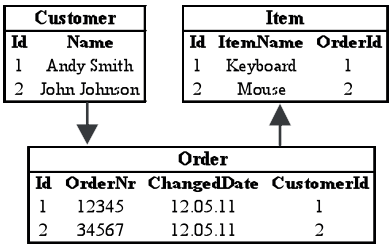


Figure 3.6: The relational entity model and initial state for program P.

Table 3.1: Test configuration for program P.

Table	Table column	Test case identifier
Order	OrderNr, ChangedDate	Customer.Name
Item	ItemName	Order.Customer.Name

Domain modeling

Prior to test execution, test data have to be prepared for the specific system component to be tested. As elaborated in Section 3.2.1 we model the input domain of the system under test as classification trees and use the models to drive the generation of test data or select production data for testing. Whether the test data is real system data or generated, the output of the test data preparation process is a test suite on which the system under test can be executed. Following our example, a classification tree model for the example program P is shown in Figure 3.7. The root node `Program P`, the classifications `Number of orders for customer`, `Item added` and `Item deleted`, and the classes (`0`, `1`, `>1`) and (`Y`, `N`) constitute the classification tree model, whereas the bottom part represent the partitions (abstract test specifications) with each line representing a partition.

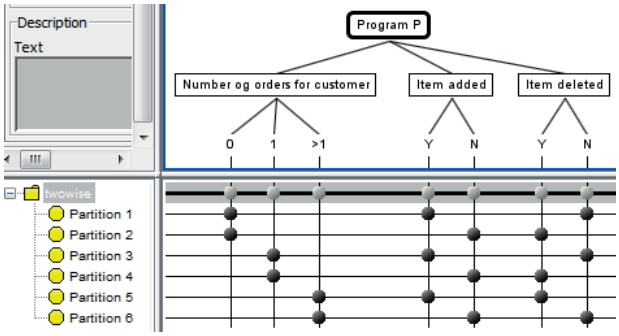


Figure 3.7: A classification tree model for program P.

From a particular test model, the tester may generate an abstract test suite specification (a set of partitions) according to a given coverage criteria (e.g. pair-wise) and then generate test data according to the specification in order to execute the tests. Alternatively, the tester could base the regression test on existing production data and match the data against the partitions in the model, in order to select tests in such a way as to maximize partition coverage. In the initial state of program P shown in Figure 3.6, there are two test cases, namely customer Andy Smith and John Johnson. DART will match the test case Andy Smith with Partition 3 as he has one order in which an item will be added, and the test case John Johnson with Partition 4 as he has one order in which an item will be deleted. When the test cases have been matched to partitions, DART selects test cases in a balanced way across partitions, and selects the most diverse ones within each partition. In the trivial example the selection is meaningless as there are only two test cases from two different partitions. However, this is important in realistic database applications, as test cases can be numerous, expensive to run, and manual inspections of deviations are time-consuming.

Test execution

During test execution DART will log all data manipulations related to the specific test configuration. The way data manipulations are recorded and logged is through dynamically generated database triggers on the tables specified in the test configuration. A trigger is procedural code that is automatically executed in response to certain events on a table or view in a database. Pseudo-code for generating the triggers is shown in Figure 3.8. As the algorithm shows, a trigger is generated for each table in the test configuration. Each of the generated table triggers is defined to insert a row into the DART log table for each data manipulation on the columns specified in the test configuration for the given table. Insert and delete operations are always done at the row level and DART will log values for all table columns in the test configuration when an insert or delete operation takes place. Update operations can be attribute specific, so DART will only log the table columns in the test configuration that is actually updated. The triggers are dynamically generated as a Data Definition Language (DDL) string, which is executed in the end to store the actual triggers in the database.

Thus, DART dynamically instruments the database of the system under test by generating test-configuration specific database triggers when the test is started. During test execution these triggers will fire on any insert, delete or update on the table columns in the test configuration and store the database operations into a DART log table. One data manipulation operation results into one row in the log table matching the format <`test case identifier`, `table name`, `column name`, `old value`, `new value`>. The `test case`

```
Input:    T is the set of tables in the configuration
          C is the set of columns from all tables in the
configuration
          T_C: 2^(T×C) is the set of (table, column) pairs
          T_R is a test run

Algorithm generateTriggers(T_C, T_R)
begin
1. for each table t ∈ T do
2.   testCaseID ← getTestCaseID(t, T_R, oldValue, newValue);
3.   triggerStatement ← "Create trigger on table t that fires after
                        insert, delete or update ";

4.   for each (t,c) ∈ T_C do
5.     triggerStatement.append("if insert operation then insert
           (testCaseNr, t, c, inserted value) into DART log table");
6.     triggerStatement.append("if update operation then
           if update on column c then insert(testCaseID,
           t, c, old value, new value) into DART log table");
7.     triggerStatement.append( "if delete operation then insert
           (testCaseID, t, c, deleted value) into DART log table");
     end loop;
   end loop;
8. execute triggerStatement;
end
```

Figure 3.8: Algorithm for trigger generation in DART.

`identifier` (e.g., the customer name) is what uniquely identifies the test case that causes the operation to be executed. It is devised on the fly according to the specification in the test configuration. `Table name` and `column name` are the names of the table and column the operation is executed on, respectively. `Old value` and `new value` refer to the values of the attribute prior to and after the operation execution, respectively. Old value is given the static value "Inserted" for insert operations, while new value is given the static value "Deleted" for delete operations. After test execution the triggers are automatically deleted from the database of the system under test.

A test run is done once with the original version of the system (baseline) and once with the changed version of the system (delta), which is subject to regression faults. If the test uses production data the database is reset to the initial (baseline) state before the delta test is run, to ensure that both runs start out with the same database state. Various mechanisms are available to reset the database, e.g. flashback to restore point. This is done by creating a restore point in the database after the test configuration is defined and the test data is prepared, but before the execution of the baseline run starts. The restore point defines the state of the database at the time it is created and will ensure consistency between the test runs. This procedure is not needed when using generated test data, as new sets of test data with identical characteristics are generated for each test run, rather than reusing the same set of test data for both the baseline and delta run.

In our example, a test run on program P, with the test configuration from Table 3.1 and

the initial state from Figure 3.6, would result in the DART log data shown in Table 3.2. For the test case Andy Smith, one insert operation and one update operation are executed, as logged in row 1 and 2 of Table 3.2, respectively. For the test case John Johnson one delete operation and one update operation are executed, as logged in row 3 and 4 of Table 3.2, respectively.

Table 3.2: Example DART log table after the baseline run.

Id	Test run id	Test case	Table name	Column name	Old value	New value
1	1	Andy Smith	Item	Item Name	Inserted	USB Stick
2	1	Andy Smith	Order	Changed Date	12.05.11	14.05.11
3	1	John Johnson	Item	Item Name	Keyboard	Deleted
4	1	John Johnson	Order	Changed Date	12.05.11	14.05.11

It turns out that program P contains a fault. The `changedDate` of the order should be updated to today's date when an order is changed but it is updated to tomorrow's date. The fault is corrected (underlined) and a new version of P, called P' is shown in Figure 3.9. For illustration purposes let us assume a regression fault in P': the update of the order in `removeItemFromOrder` method is completely removed, rather than fixed (line struck through). After resetting the database into the same initial state as before the first test run, the test is executed again on the changed program version P'.

After both test runs, the DART log table contains the information shown in Table 3.3. Three additional rows are logged for the delta run: An insert and an update operation for the test case Andy Smith in row 5 and 6, and a delete operation for the test case John Johnson in row 7.

Table 3.3: Example DART log table after both test runs are executed.

Id	Test run id	Test case	Table name	Column name	Old value	New value
1	1	Andy Smith	Item	Item Name	Inserted	USB Stick
2	1	Andy Smith	Order	Changed Date	12.05.11	14.05.11
3	1	John Johnson	Item	Item Name	Keyboard	Deleted
4	1	John Johnson	Order	Changed Date	12.05.11	14.05.11
5	2	Andy Smith	Item	Item Name	Inserted	USB Stick
6	2	Andy Smith	Order	Changed Date	12.05.11	13.05.11
7	2	John Johnson	Item	Item Name	Keyboard	Deleted

Test Analysis

After a test is executed on two different versions of the SUT, the two test runs are compared with each other. The output of the test execution is a DART log table filled with all data manipulation operations of the respective test runs. The comparison uses the SQL set operations minus and union to compute the difference between the two runs, as follows:

Program P'

```
void removeItemFromOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                    where Order.orderNr = orderNr;
  if(orderId != null) {
    delete from Item where Item.itemName = itemName and
    Item.orderId = orderId;
    update Order set Order.changedDate = tomorrow
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void addItemToOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                    where Order.orderNr = orderNr;
  if(orderId != null)
    insert into Item (ItemName, OrderId) values ( itemName
orderId);
    update Order set Order.changedDate = today
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void main() {
  addItemToOrder(12345, "USB stick");
  removeItemFromOrder(34567, "Mouse");
}
```

Figure 3.9: The example program P', which is a modified version of program P.

<Log data from baseline> MINUS <Log data from delta>
UNION ALL
<Log data from delta> MINUS <Log data from baseline>

The comparison operation will reveal all differences between the baseline and delta runs with respect to the test configuration. The deviations, grouped by the test case identifier, are presented to the tester, which in turn has to determine whether the deviation is a regression fault or not.

In our example the output of the test is the deviations between the two runs, as shown in Table 3.4. There is one deviation due to the changed update in `addItemToOrder` (row 1-2) and one deviation due to the missing update in the delta version of `removeItemFromOrder` (row 3). By analyzing the deviations in Table 3.4, the test engineer can verify that the change in test case Andy Smith is due to correct changes in P', whereas the missing update in the test case John Johnson is due to a regression fault.

As the baseline run essentially serves as the test oracle, DART will identify regression faults introduced in the delta version of the system, but will not identify faults that are

present in both the baseline and delta run. In practice, the same baseline is used for testing several consecutive deltas. After each test, the deviations that are correct in the delta are updated into the baseline. Thus, the baseline is continually improved and the test oracle increasingly more accurate.

Table 3.4: The deviations between the test runs for P and P'.

Id	Test case	Table name	Column name	Old value	New value	Test run
1	Andy Smith	Order	Changed Date	12.05.11	14.05.11	Baseline
2	Andy Smith	Order	Changed Date	12.05.11	13.05.11	Delta
3	John Johnson	Order	Changed Date	12.05.11	14.05.11	Baseline

Even with efficient test selection, a regression test may result in numerous deviations, which need to be analyzed to determine their cause. Further, many deviations are likely to be the result of the same regression fault or the effect of the same change. In order for the tester to handle the regression test deviations efficiently, DART clusters the deviations based on their output characteristics. The goal of the deviation clustering process is to identify groups of deviations caused by identical changes or regression faults. Then the tester can prioritize deviations in such a way as to cover at least one deviation from each cluster, and thus maximize fault detection if the inspection budget is limited. The full details of deviation clustering is provided in Paper 3 (Chapter 3 in Part 2).

3.2.3 Summary

In summary, we then have an approach to regression testing of database applications that includes the following characteristics:

- Automatically observes and compares executions of the system under test, in order to identify regression test deviations.
- Partially automates verification of test cases and groups regression test deviations according to their output characteristics for efficient test analysis.
- Takes a systematic approach to control the variation in the input domain by making use of classification tree models for black-box specification of the input domain of the system under test.
- Automatically generates test specifications from test models
- Automatically generates test data that is needed to execute test cases, or alternatively selects production data, both according to the test specifications from test models.

Chapter 4

Research Methodology

This thesis reports on practice-driven research aimed at finding applicable solutions to test challenges derived from engineering practice. In our context, the identification of such challenges and the evaluation of proposed solutions were performed in collaboration with The Norwegian Tax Department [35]. The research methodology applied for this thesis included understanding the industrial problems in context, assessing the existing literature in terms of its match to the defined problems, proposing solutions to the test challenges and develop a tool to deploy the solutions within the application context, conducting empirical studies to evaluate the proposed solutions, and iteratively improving the test methodology and tool based on the results of the empirical studies.

4.1 Understanding the practical problems

Because this is practice-driven research, we started the work by understanding the application context in order to identify and carefully define the main problems with engineers. To ensure a close and fruitful collaboration, I was located on-site at our partner's premises throughout the project and we were given access to the necessary artifacts to help us understand the context and test challenges. During initial contact, they reported that their system had grown to a size and complexity where the current, manual testing processes were no longer deemed to be adequate or cost-effective. They both felt the need to, and were curious to introduce a higher degree of test automation in order for their testing regime to scale better. Through further investigation of system documentation and discussions with project staff members at different levels, it became evident that a particular area of concern was the vast number of regression faults, occurring as a result of unanticipated ripple effects from changes in the system. This indicated the need for implementing systematic, automated regression testing in the project.

Their system portfolio mainly consisted of large database applications, which were

dependent on several core batch processes to carry out the centralized business logic guided by taxation laws and rules. However, these batch processes were automated and consisted of complex programs that process very large numbers of transactions and span a large input data space. Consequently, testing suffered from the following challenges: manually verifying programs with such a variety of input combinations (large number of test cases) does not scale, they needed a way to determine which test cases to execute to achieve satisfactory regression testing, and they needed effective ways to procure the test data necessary to execute test cases in order to promote test automation.

After understanding the industrial context and test challenges in the context of database applications, we assessed the existing literature in order to learn the status of related work. In general, the literature pertaining to the testing of database applications is sparse and there exists little research on how to test database systems in a systematic and cost-effective manner. Most literature in the field is aimed at assessing performance of database management systems rather than testing the database application system for functional correctness, not to mention regression testing. Furthermore, there are few empirical studies in industrial settings. As mentioned in Section 2.2, the two most relevant initiatives were the AGENDA Framework [8, 7, 9, 12] and the SIKOSA project [17, 4, 18]. However, the AGENDA framework was not intended for regression testing and it had only been evaluated on smaller academic examples. We found the ideas hard to scale to the needs of our partners. Some of the ideas from the SIKOSA project aligned with our test challenges and thoughts regarding possible solutions. However, apart from some experimental performance measures, they did not refer to any evaluations regarding fault-detection capabilities or cost-effectiveness, let alone in a realistic setting. Neither did any of them refer to available tools to support testing in practice.

4.2 Develop regression test methodology and tool

As a result of the problem identification and the lack of existing solutions to address this problem, we developed a novel and practical regression test methodology and tool. It was important for the proposed solution to not only address the immediate test challenges of our partner, but also ensure its applicability in a broader test context, namely regression testing of database applications in general. The proposed regression test methodology and tool are described in Section 3.2. The tax department contributed a system developer to the project in order to help implement the test methodology in a tool suited for their application context.

4.3 Empirical studies

A fundamental part of this thesis is the execution of empirical studies in a real and representative database application development environment. We conducted several empirical studies and experiments to evaluate our test solutions and iteratively improve the methodology and tool for regression testing. First, we evaluated the fault detection capabilities of our regression test solution through a case study in the tax department environment. Then we conducted a large scale experiment to evaluate different ways of conducting test case selection based on classification tree models, before doing further experiments on the effect of clustering regression test deviations for improved inspection efficiency. Last, we conducted a large scale case study comparing strategies for regression testing, both regarding the construction of test suites and the setup of test data needed to execute the regression tests.

When comparing techniques, e.g. test case selection techniques or deviation clustering strategies, we have used statistical tests to determine whether there exists a significant difference. Because our data samples do not fulfill the underlying assumption of normality and equal variance, we have used non-parametric statistical tests [1, 36]. More specifically, we have used two-tailed Mann-Whitney U-tests (Wilcox test in R) to conduct pair-wise algorithm comparisons. In general, we have used $\alpha = 0.05$ whenever referring to statistical significance, but have also reported the exact p-values for each comparison made, in order for the results of the statistical tests to be transparent for the readers.

Because statistical tests only address statistical significance, we have also used the $\hat{A}_{12}$ effect size measure to assess the practical significance of the differences. When comparing sample A to sample B, an $\hat{A}_{12}$ effect size measurement value of 0.5 indicates that there is no difference between the two samples, whereas values above 0.5 indicates that sample A is superior to sample B, and the opposite for values smaller than 0.5. The further away from 0.5, the larger the effect size. We have reported the value of the effect size measurement, but for increased legibility we have also classified the effect size into Small, Medium and Large categories. The categories are based on the following standard intervals: Small $<$ 0.10, 0.10 $<$ Medium $<$ 0.17 and Large $>$ 0.17, the value being the distance from the 0.5 [36].

To account for randomness in sampling we repeated the exercises between 100 and 1.000 times depending on the amount of time taken for each execution and the number of comparisons that needed to be carried out. This was done in order to ensure robustness of the statistical results reported.

Chapter 5

Summary of Results

In this section, the key results of the papers submitted as part of this thesis are summarized.

5.1 Evaluation of fault detection capabilities

Paper 1: Industrial Experiences with Automated Regression Testing of a Legacy Database Application

This paper presents our proposed solution for functional regression testing of database applications and reports an evaluation of the tool implementation (DART) in a real application setting. The approach to regression testing is elaborated in detail in Chapter 3. Thus, this section will focus on the results from evaluating the approach and tool, when applied for regression testing of business critical batch programs in the tax department of Norway.

The evaluation showed that the approach and tool lead to high fault detection capabilities. As an early pilot evaluation of DART, we re-tested a previous system release, which had already undergone the regular, manual testing and QA activities. Five regression faults had been identified during the regular testing routines in the project, and five additional regression faults had been discovered in the production environment after it was released. The purpose of the pilot evaluation was to investigate whether we could identify the same ten regression faults, and possibly additional, undiscovered faults, with the DART tool.

The results showed that DART revealed eight of the ten regression faults that were previously found during testing and operation. The two previously detected faults missed by DART were not found due to the insufficient coverage of the test suites; none of the test cases exercised the two faulty situations. The pilot test was based on a random

selection of test data, and the missing coverage suggested the need for more systematic approaches for selecting test cases in order to ensure more complete testing. However, the pilot evaluation showed that the tests executed with DART also discovered nine previously undetected faults, that is nine faults that were still present in the production system and needed to be corrected. These faults were registered as one "A defect", seven "B defects" and one "C defect" on a criticality scale ranging from A to C, where A is the most critical one. In total the test with DART uncovered 17 distinct faults, which is a substantial improvement in terms of fault detection capability.

Furthermore, DART was used to support regression testing of batch jobs in the core functional areas of the SOFIE application for eight subsequent releases. DART was then used as a supplement to manual testing, not as a replacement. That is, DART was used as final verification of the releases after the acceptance test was finished and the release was ready to ship. A total of 37 faults were detected by regular testing during these eight releases, while an additional 22 faults were detected by DART. DART then helped uncover more than a third of the defects found during regression testing of the subject system during these eight releases. Put in other words DART helped identify approximately 60% more regression faults than what would have been detected without it. We consider this to be of substantial impact, especially since DART was only used as a "last check".

As part of the evaluation, we also looked at defects reported from the production environment of the system. We learned that even when combining manual testing with DART, some faults still slipped through to production. We found a total of 14 faults reported within the functional areas we had been testing. We went through the defects to understand why these faults were not detected through testing with DART prior to being released. The detailed analysis showed that six of them had actually been found and reported by DART, but there were not enough time to fix them prior to release. Then they had also been reported from the production environment before they got fixed. In addition, there were a few defects on performance issues specific to the production environment and one fault that was missed as we did not execute that part of the functional domain in that particular test round. Most importantly, there were three faults that were not discovered due to insufficient coverage of the test cases executed. This suggested the need for generating test data in order to ensure test suites with predictable and more complete coverage, which was later on addressed and reported in paper 4.

Another thing we learned from this study was the need for a systematic approach to test case selection in order to reduce the number of redundant test cases executed. This is important for the regression test approach to scale to large systems by reducing the time spent on analyzing regression test deviations. This is addressed in detail in paper 2,

but initial results reported in this paper already showed significant potential savings in inspection effort when selecting test cases systematically according to partition coverage of classification tree models.

5.2 Evaluation of test case selection techniques

Paper 2: Test Case Selection for Black-Box Regression Testing of Database Applications

The purpose of the study undertaken in this paper was to address a core challenge in regression testing, namely selecting test cases. This problem is particularly acute when test results must be checked manually or execution costs are large, and observations from initial regression tests on randomly selected test data indicated significant redundancy, leading to large inspection costs. In order to reduce test costs and for the regression test approach to scale properly for testing large database applications, we thus needed a refined way of selecting regression test cases. Hence, the objective was to select test cases in such a way as to minimize regression testing effort while retaining maximum fault detection power.

Our black-box approach to regression testing is based on classification tree models, which are used to model the input domain of the system under test. Therefore, this study investigates strategies for selecting regression test cases based on classification tree models. Such models are used to partition the input domain of the system being tested, which in turn are used to select and generate system test cases so as to achieve certain strategies for partition coverage. To refine regression test selection, we combined similarity-based test case selection with such a partition-based approach, so that test cases can be selected within partitions while maximizing diversity among them.

An experiment was conducted to evaluate various selection strategies, including a pure similarity-based selection and partition-based selection incorporating both a random and similarity-based selection strategy within partitions. Both fault detection rate and selection execution time were assessed. For the similarity-based approaches, several similarity functions and selection algorithms were investigated. The study addressed the following research questions (the details of the algorithms and the selection strategies are explained in Section 3 and 4 in Paper 2):

RQ1 For each selection algorithm, which similarity function, i.e. Euclidian, Manhattan, Mahalanobis and NCD, is best suited for defining similarity between test cases, in order to obtain the best fault detection rate and selection execution time when

performing similarity-based selection of test cases generated from classification tree models?

RQ2 When used in combination with their best similarity function (RQ1), which one of the selection algorithms, i.e., evolutionary and greedy, provides better fault detection rates and selection execution times, when selecting test cases generated from classification tree models?

RQ3 Which one of the selection strategies, random partition-based, similarity partition-based and pure similarity-based (the two latter ones incorporating the best combination of similarity function —RQ1—and selection algorithm—RQ2) provides better fault detection rates and selection execution times when selecting test cases generated from classification tree models?

RQ4 Comparing the best selection approach from RQ3 and a random approach, which one provides better fault detection rates and selection execution times when selecting test cases generated from classification tree models?

The results showed that applying a partition-based selection, i.e. select model partitions in a balanced way, and then select test cases within each partition either at random or by maximizing diversity between them, provided a dramatic improvement over random test case selection. For example, by selecting 5% of the test cases in a test suite, the fault detection rate of partition-based selection is nearly 80%, as opposed to 25% for random selection. Finding nearly 80% of the faults at the cost of only 5% of the test suite yields highly significant savings in practice, as it both reduces test execution time and more importantly reduces the manual effort associated with analyzing test results.

Whether to opt for a similarity partition-based selection or a random partition-based selection was not clear cut as their differences where marginal. The selection execution time favored random partition-based, with significant differences for all sample sizes. However, for smaller sample sizes, which are of main interest here, the difference is less than a few minutes, i.e. for sample values up to 50%, execution time ranges from near 0 to up to 4 min for similarity partition-based, as opposed to less than a second for random partition-based. These differences in selection execution time have limited consequences in practice. In terms of the fault detection rate, similarity partition-based was better than random partition-based for all sample sizes, though with modest improvements. A detailed analysis concluded that the similarity partition-based selection strategy should be applied when a large number of test cases are contained in each partition and there is significant variability within partitions. If these conditions are not present, incorporating similarity measures is not worthwhile, since the gain is negligible over a random selection within each partition.

As part of the study we investigated several similarity functions and selection algorithms to identify the most effective similarity-based selection strategy in our context. The Mahalanobis similarity function combined with the evolutionary selection algorithm proved to be the most efficient with regards to fault detection rate. The fact that accounting for correlation is the main differentiating factor between the Mahalanobis similarity function and the two simpler similarity functions (Euclidian and Manhattan) could indicate that it is important to consider correlation among model properties. The NCD similarity function fell short of the others in terms of fault detection rate. The most plausible reason is that the test cases are represented by a fairly simple structure that accounts for very little information distance between their compressed versions. Thus, NCD is not able to pick up the minor differences as well as simpler geometrical functions. The evolutionary selection algorithm provided better fault detection rates than the greedy selection algorithm for all selection samples sizes except for 5%, where there was no statistical significance. For all other sample sizes, the difference was notable (large effect size). Evolutionary also converges to 100% fault detection much faster than the greedy approach, i.e. it reaches 98.5% and 100% detection rates at 60% and 80% sample sizes, respectively, whereas greedy does not reach a 100% detection rate until the sample size is 100%.

5.3 More effective regression test analysis by clustering deviations

Paper 3: Clustering Deviations for Black Box Regression Testing of Database Applications

This study investigates how to cope with the many discrepancies (deviations) that can be observed when running regression test cases on a new version of a system. In other words, how can we help testers analyze such deviations effectively and decide whether they are symptoms of regression faults or the logical result of changes. A problem that is highly important in practice, but one that has not been given much attention in the research literature so far. Our regression test approach enables effective execution and automatic comparisons of large number of test cases and pin-points exactly which test cases that result in deviations and thus are candidates for regression faults. Yet, manual test effort is required in order to inspect the deviations and decide their cause. There may be numerous deviations from an individual regression test, and although effective test case selection reduces the number, there may occasionally still be more than a tester can effectively handle.

As the number of deviations tend to be far greater than the number of unique changes or regression faults, our aim was to group deviations resulting from the same change or regression fault, in order for the tester to inspect as few deviations as possible while still remaining confident of finding all regression faults. We chose to base our solution for grouping deviations on clustering, that is algorithms which aim at discovering groups and structures in data in such a way as to maximize the distance between groups, and minimize the distance within groups.

The study addressed the following research questions:

RQ1 Can clustering serve as an automated, accurate grouping strategy for grouping regression test deviations?

RQ2 What kind of input data to the clustering process yields the most accurate grouping of the deviations?

RQ3 How much effort can the tester be expected to save in regression test and deviation analysis, when systematically grouping test deviations prior to analysis, as opposed to the current random inspection?

The results showed that clustering indeed can serve as an accurate strategy for grouping regression test deviations. We measured the level of entropy across both clusters and groups of deviations in order to determine the accuracy of the clustering. Entropy gave us a normalized measure of the purity of each cluster, and the spread of each type of deviation across clusters. A perfect clustering would have as many clusters as there were regression faults and changes, and each cluster would only contain deviations matching a unique fault or change, thus yielding an entropy of zero. In practice, it is more important to obtain zero cluster entropy than zero deviation entropy, as this would ensure that all the clusters are homogenous (only matching one regression fault or change). Under such circumstances, the tester would always conduct a complete deviation analysis when inspecting one deviation from each group, only with the potential risk of having to analyze a few more deviations than necessary if the deviation entropy is not perfect.

The case study, conducted in a real development setting, evaluated the proposed clustering strategy across four different regression test campaigns covering three different parts of the system under test. For three out of the four test campaigns, clustering accuracy was perfect for at least one type of input. Additionally, four types of input yielded homogenous clusters for all test campaigns. In terms of inspection effort, this means that for the four test campaigns under study, and when using a one-per-cluster sampling strategy, it was estimated that testers would only have to analyze a very small percentage of deviations while still covering all regression faults and changes in the regression tests. The specific number of deviations to inspect were three out of 48, two out of 47, two out

of 84, and five out of 43, respectively. This is a dramatic increase in efficiency for deviation analysis in regression testing, and a way to achieve much higher confidence when analyzing deviations under time constraints. Such results are more specifically important for scaling regression testing to large database applications.

The case study also made a thorough assessment of the kind of clustering input that yielded the most accurate results in our context, where test cases are derived from classification tree models. The sources of information used as clustering input per test case were properties from the test models and various aspects of the deviation output, i.e. which tables and table columns that deviated along with the database operation causing the deviation (inserts, updates, or deletes). In general, nearly all clustering results based on model properties yield poor results, indicating they are not suited to cluster deviations. A plausible explanation is that the model describes the input domain of the regression test rather than the output of the test. Although the two are related, the relationship is complex, and the results suggest that the information available in the deviation output is more relevant to clustering the deviations. Overall, the information that led to the most accurate deviation clustering was the combination of columns and operations in the deviation output, indicating differences in columns and operations being manipulated across versions for a given test case. That specific combination of input resulted in perfect accuracy for two of the test campaigns, while achieving perfectly homogeneous clusters for the two others, with a limited degree of dispersion across more than one cluster for common-cause deviations.

5.4 Effective regression test strategies for database applications

Paper 4: Cost-effective Strategies for the Regression Testing of Database Applications: Case study and Lessons Learned

This study was undertaken in order to assess the practice of using production data for testing database applications and evaluate the alternative practice of using generated test data following various strategies. The quality of the production data is indisputable, however using production data for test automation purposes offers some challenges. They may be rigid to procure, confidentiality issues have to be sorted, and the test coverage of a given set of production data is unpredictable and may not satisfy test specifications. Generated test data address all those flaws, but at a potentially high initial cost of setting up the mechanism to generate test data. Also, which test strategy to embrace when generating test data in order to ensure the most exhaustive regression testing possible is

not obvious.

We adopted a combinatorial testing strategy based on our classification tree models, which is an attractive strategy for generating compact n-way test suite specifications. During initial tests on combinatorial test suites we found quite a few faults that bore little relevance to the practical operation of the system under test. While there is no harm in finding faults regarded less important, as long as the more important ones are still detected, it made us question whether combinatorial test suites were adequate to support regression testing. Thus, we made an analysis of the operational profile of the system under test, which was used both to check how well the combinatorial test suites aligned with the operational profile, and as an alternative strategy to drive generation of synthetic test data. We assessed the adequacy of these alternative strategies for generating synthetic test data (various combinatorial test strategies and operational profile testing) in terms of revealing important faults, along with expected coverage and fault detection capabilities of tests on production data. The working hypothesis was that the combination of combinatorial tests and operational profile tests on generated test data would prove to be a good alternative to using production data and ensure more effective and complete regression tests.

The following research questions were addressed in the study:

RQ1 Given the common practice of using production data for (regression) testing database applications, its analysis and assessment is of high practical interest:

1. What level of partition coverage and model property coverage can you expect from production data used for testing, when matched against a realistic classification tree model?
2. How much redundancy can you expect given the same tree, i.e. do many instances of test data cover the same partitions?

RQ2 Given that combinatorial testing is one of the major testing approaches recommended in the literature, better understanding what it could achieve in database application testing and its drawbacks is of high practical interest. When comparing pair-wise, three-wise, and weighed pair-wise combinatorial test suite specifications with the test suite specifications based on the system operational profile, how well do they align, i.e. how representative are combinatorial test suite specifications of the system usage?

RQ3 Ultimately, we want to favor a test strategy that reveals important faults. By important, we mean a high relative likelihood of occurrence during system operation, e.g. the number of taxpayers potentially being harmed by the fault. Given the

following types of test suites: (1) synthetic pair-wise (2W), (2) synthetic three-wise (3W), (3) synthetic pair-wise weighed according to the operational profile (W2W), (4) synthetic and based on the operational profile only (OP), and (5) based on production data only (PD), which test strategy does achieve higher fault detection and best reveal important faults in the context of regression testing?

RQ4 Can combinatorial testing be effectively combined with testing based on operational profiles or production data? Intuitively, these techniques can be expected to target different faults. Are they complementary, and if so, what combination yields the best fault detection rate?

The results from a case study in our application environment showed that combinatorial testing strategies are effective, both in terms of the number of regression faults discovered, but also in terms of the importance of these faults. This is despite the fact that the combinatorial test suite specification shares very little with test suite specifications derived from the operational profile. In fact, the combinatorial tests detected all the faults that were detected using operational profile tests and many more. A plausible explanation why this is the case is that faults can be triggered by the interactions of a few equivalence classes. So despite being unrealistic (their partitions have low probability of occurrence), combinatorial test case specifications still seem to have the ability to detect important faults.

The tests on production data revealed fewer faults than the combinatorial tests, but captured faults that neither of the combinatorial nor operational profile tests on generated test data were able to capture, and on average the faults were also more important. We found this to be due to the fact that the test models from which the generated test data are derived were missing important properties, which in turn shows that modeling a large input domain is hard even with domain experts involved. Thus, the combination of combinatorial tests and operational profile tests on generated test data are not sufficient to adequately regression test the system as ignoring tests on production data could see important faults slip through.

As regression testing is more effort intensive with production data, we recommend running continual regression tests using the most cost-effective, generated combinatorial test data (pair-wise) throughout the system release development, e.g. after each code check-in (defect correction). Then, when reaching a stable system release, we advise to set up and run additional regression tests based on production data (carefully selected according to partition coverage) in order to capture additional relevant faults. This would both ensure quick feedback for the developers after defect corrections, at relatively low cost, but also the execution of complementary regression tests to detect faults outside the scope of the test model, prior to system delivery. In practice, test models should also be

iteratively improved when possible, when shortcomings are being exposed by production data testing. The generated test data would then continuously improve and, in the long run, this would reduce the need for production data testing and decrease the cost of regression testing.

Chapter 6

Directions for future work

Our proposed regression test approach introduces automation into several parts of the regression test procedure, but is not fully automated. As long as there does not exist a complete and precise test oracle, there will always be manual labour associated with the analysis of test results. Careful measures to select or prioritize regression test cases help reduce the test effort, as do measures taken to group regression test deviations according to their cause. These measures are necessary in order for regression testing to scale to large database applications, but still some manual effort is required for test verification. In order to ensure a completely automated regression test process, further research would be required on how to effectively build a precise test oracle that scale for industry size applications. However, until someone walks that extra mile, more efforts could be taken in order to further increase the scalability of regression testing and help reduce manual work. One thing that has not been part of this work, but nevertheless would have been very relevant is an automated way of predicting the expected changes to the result sets based on the changes made in the new system version. A precise mechanism for doing so would ensure that expected changes could be automatically detected and verified in a regression test and consequently eliminate all deviations caused by these changes from the resulting set of regression test deviations. Then, the remaining deviations would result from regression faults.

Another relevant future addition, which is not research per se, but rather technology transfer, would be to implement an open-source platform-independent tool variant of the regression test approach suggested in this thesis. Our ambitions was set at doing so, but unfortunately this process often goes beyond the scope a research project and did, for various reasons, not materialize. The regression test tool we have developed is sophisticated and applicable in an industry setting, but nevertheless developed in our specific context and is neither open source, nor platform-independent.

Chapter 7

Concluding Remarks

Regression testing is highly important in order to maintain system quality when new versions of a system are released. Regression testing is also expensive as it attempts to exercise the system under test in a comprehensive manner. By nature, regression tests are repetitive and thus well suited for test automation. However, automating regression testing is in general not straightforward, at least not for higher level system testing. In this thesis, we have focused on test automation in the context of functional, black-box system level regression testing of database applications. In such a context, core challenges regarding regression testing are (1) how to collect a test baseline and compare test results across system versions given the fact that it is difficult to build precise test oracles for database applications with many highly complex queries, (2) how to reduce test effort by selecting or prioritizing test cases, (3) how to cope with the many discrepancies that can be observed from regression tests across system versions, and (4) how to go about regression testing in practice, namely what test strategies to opt for and how to effectively provide test data needed for test execution.

As a first step towards addressing the above challenges, we proposed in Paper 1, a methodology and tool prototype for regression testing of batch programs running in large database applications and evaluated the fault detection effectiveness of the approach in a large application with real faults. When the approach was applied to regression test a past system release, it detected eight of the ten faults previously detected during testing and operation, but also revealed an additional nine faults that were still present in the operation of the system. Furthermore, when applied as a last regression test prior to eight consecutive system releases, the approach helped identify an additional 60% regression faults to those found during regular testing, which otherwise would have been released. We believe this is a substantial contribution in terms of increased fault detection that helps mitigate risk when releasing new versions of the system, as it is more thoroughly, yet efficiently tested, causing less regression faults to be released.

The next step was to further improve the scalability of the regression test approach, which was addressed in two dissimilar, yet complementary ways in Papers 2 and 3. First, we addressed a core challenge in the input part of regression testing, namely selecting test cases. Initial regression tests on randomly selected test cases indicated a lot of redundancy, i.e. many test cases resulting in the same regression faults. In order to reduce test costs and to improve the scalability of the regression test approach, we needed a refined way of selecting regression test cases. We made use of classification trees to model the input domain of the system under test, which provided a structured way of specifying test cases. When using production data as basis for regression testing, we could then match the data against the test case specifications, in order to learn their distribution across model partitions. Then, we could apply a partition-based test case selection to select test cases in a balanced way from the model partitions covered, and then either selects test cases at random within each partition or, alternatively, based on diversity within each partition. The latter is effective when the number of test cases in partitions is large and the equivalence classes are defined in such a way that they contain significant variation. Such partition-based selection proved very efficient in our context, where for example selecting only 5% of the test cases in a test suite helped reveal nearly 80% of the faults, thus significantly reducing test effort while still remaining confident about finding most faults. Additionally, we have implemented a solution for automatically generating test data according to test suite specifications from classification tree models. This enables us to use the classification tree models to automatically generate test suite specifications according to combinatorial selection criteria, e.g. pair-wise and three-wise, and in turn automatically generate test data needed to execute the test cases. Thus, we have effective test selection methods for both situations here we use production data and generated data for regression testing, both based on classification tree models.

The second attempt to help scale regression testing to large database applications was taken in paper 3. Here, we aim at addressing another core challenge in regression testing, namely how to effectively handle the many discrepancies that can be observed when running regression test cases on a new version of a system. In other words, how can we help testers analyze such deviations, which can be due to either changes or regression faults, and decide what are their actual causes. We made use of clustering to help group regression test deviations based on patterns in their output characteristics. The goal was to identify groups of deviations caused by the same changes or regression faults. Our clustering approach to regression test deviations showed encouraging results. Among the four test campaigns assessed, deviations were clustered perfectly for two of them, while for the other two, the clusters were all homogenous, i.e. all deviations in a cluster are results of the same change(s) or fault(s). Because, we always achieved homogenous clusters, the

testers would be certain to cover all groups of regression faults by inspecting one deviation from each cluster, which could yield significant savings in inspection effort. For example, in our test campaigns that resulted in inspecting three out of 48, two out of 47, two out of 84, and five out of 43 regression test deviations, respectively.

Then we conducted a case study assessing, comparing, and combining various strategies for regression testing. Our observations of regression testing in a real development environment suggested that combinatorial testing alone was not sufficient to capture all kinds of regression faults, and should therefore be supplemented with regression tests better aligned with the operational profile of the system under test, as intuitively they target different faults. One way to go about operational profile testing is to rely on production data, as by nature production data will in general be representative of the system operation. However, as production data is not optimal when trying to automate regression testing, a more attractive strategy would be to use operational profiles to drive the generation of operational profile test suites from our classification tree models, and then automatically generate test data according to the operational profile test suite specifications. The case study showed some interesting results, though. First of all, the combinatorial tests proved to be very efficient in terms of detecting faults and pair-wise was the most cost-effective strategy among them in our context. Second, the operational profile tests on generated data derived from the classification tree models did not offer any improvement in terms of capturing additional faults to those revealed by the combinatorial test strategies. Third, what did yield improvements was regression tests based on production data. Thus, the combination of pair-wise combinatorial testing and tests relying on production data, selected in a systematic manner according to model partition coverage, turned out to be a convincing test strategy in terms of covering a broad range of regression faults at a reasonable cost.

References for the Summary

[1] Andrea Arcuri and Lionel Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In: *Software Engineering (ICSE), 2011 33rd International Conference on*. 2011, pp. 1–10.

[2] Boris Beizer. *Software Testing Techniques (2Nd Ed.)* New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[3] John Bible, Gregg Rothermel, and David S. Rosenblum. "A Comparative Study of Coarse- and Fine-grained Safe Regression Test-selection Techniques." In: *ACM Trans. Softw. Eng. Methodol.* 10.2 (Apr. 2001), pp. 149–183.

[4] Carsten Binnig, Donald Kossmann, and Eric Lo. "Testing Database Applications." In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 739–741.

[5] Rex Black. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. New York, NY, USA: John Wiley & Sons, Inc., 2007. ISBN: 0470127902.

[6] Donald K. Burleson. *Oracle SAP Administration*. Ed. by Gigi Estabrook. 1st. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999. ISBN: 156592696X.

[7] David Chays and Yuetang Deng. "Demonstration of AGENDA tool set for testing relational database applications." In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. 2003, pp. 802–803.

[8] David Chays et al. "A Framework for Testing Database Applications." In: *SIGSOFT Softw. Eng. Notes* 25.5 (Aug. 2000), pp. 147–157.

[9] David Chays et al. "An AGENDA for Testing Relational Database Applications: Research Articles." In: *Software Testing, Verification and Reliability* 14.1 (Mar. 2004), pp. 17–44.

[10] David M. Cohen et al. "The AETG system: an approach to testing based on combinatorial design." In: *Software Engineering, IEEE Transactions on* 23.7 (1997), pp. 437–444.

[11] Myra B. Cohen et al. “Constructing Test Suites for Interaction Testing.” In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE ’03. Portland, Oregon: IEEE Computer Society, 2003, pp. 38–48.

[12] Yuetang Deng, Phyllis Frankl, and David Chays. “Testing database transactions with AGENDA.” In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. 2005, pp. 78–87.

[13] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. “Test Case Prioritization: A Family of Empirical Studies.” In: *IEEE Trans. Softw. Eng.* 28.2 (Feb. 2002), pp. 159–182.

[14] Francisco G. Oliveira Neto Emanuela G. Cartaxo Patrícia D. L. Machado. “On the use of a similarity function for test case selection in the context of model-based testing.” In: *Software Testing, Verification and Reliability* 21 (2 2011), pp. 75–100.

[15] Colleen Graham et al. *Market Share: All Software Markets, Worldwide, 2013*. 2014. URL: `https://www.gartner.com/doc/2695617` (visited on 06/15/2014).

[16] Todd L. Graves et al. “An Empirical Study of Regression Test Selection Techniques.” In: *ACM Trans. Softw. Eng. Methodol.* 10.2 (Apr. 2001), pp. 184–208.

[17] Florian Haftmann, Donald Kossmann, and Er Kreutz. “Efficient regression tests for database applications.” In: *In Conference on Innovative Data Systems Research (CIDR*. 2005, pp. 95–106.

[18] Florian Haftmann, Donald Kossmann, and Eric Lo. “A Framework for Efficient Regression Tests on Database Applications.” In: *The VLDB Journal* 16.1 (Jan. 2007), pp. 145–164.

[19] Mary Jean Harrold and Alessandro Orso. “Retesting software during development and maintenance.” In: *Frontiers of Software Maintenance, 2008. FoSM 2008.* 2008, pp. 99–108.

[20] A. S. Hedayat, N. J. A. Sloane, and John Stufken. *Orthogonal Arrays: Theory and Applications*. 1st. New York: Springer, 1999.

[21] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. “Achieving Scalable Model-based Testing Through Test Case Diversity.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.1 (Mar. 2013), 6:1–6:42.

[22] Michael Hoding et al. *SAP Database Administration with Oracle*. SAP PRESS, 2008. ISBN: 1592291201, 9781592291205.

[23] Bogdan Korel, Luay H. Tahat, and Mark Harman. “Test prioritization using system models.” In: *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*. 2005, pp. 559–568.

[24] D. Richard Kuhn and Michael J. Reilly. “An investigation of the applicability of design of experiments to software testing.” In: *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. 2002, pp. 91–95.

[25] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr. “Software Fault Interactions and Implications for Software Testing.” In: *IEEE Trans. Softw. Eng.* 30.6 (June 2004), pp. 418–421.

[26] Eckard Lehmann and Joachim Wegener. “Test Case Design by Means of the CTE-XL.” In: *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*. 2000.

[27] Hareton K.N. Leung and Lee White. “Insights into regression testing [software testing].” In: *Software Maintenance, 1989., Proceedings., Conference on*. 1989, pp. 60–69.

[28] Robert Mandl. “Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing.” In: *Commun. ACM* 28.10 (Oct. 1985), pp. 1054–1058.

[29] Daniel Di Nardo et al. “Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system.” In: *Software Testing, Verification and Reliability* 25.4 (2015), pp. 371–396.

[30] Thomas Joseph Ostrand and Marc J. Balcer. “The category-partition method for specifying and generating fuctional tests.” In: *Communications of the ACM* 31.6 (1988), pp. 676–686.

[31] David S. Rosenblum and Elaine J. Weyuker. “Lessons Learned from a Regression Testing Case Study.” In: *Empirical Softw. Engg.* 2.2 (Feb. 1997), pp. 188–191.

[32] Gregg Rothermel et al. “An empirical study of the effects of minimization on the fault detection capabilities of test suites.” In: *Software Maintenance, 1998. Proceedings., International Conference on*. 1998, pp. 34–43.

[33] Patrick J. Schroeder, Pankaj Bolaki, and Vijayram Gopu. “Comparing the fault detection effectiveness of n-way and random test suites.” In: *Empirical Software Engineering, 2004. ISESE ’04. Proceedings. 2004 International Symposium on*. 2004, pp. 49–59.

[34] Patrick J. Schroeder and Bogdan Korel. “Black-box Test Reduction Using Input-output Analysis.” In: *SIGSOFT Softw. Eng. Notes* 25.5 (Aug. 2000), pp. 173–177.

[35] *Skatteetaten*. 2015. URL: `http://www.skatteetaten.no`.

[36] András Vargha and Harold D. Delaney. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong.” In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[37] Filippos I. Vokolos and Phyllis G. Frankl. “Empirical Evaluation of the Textual Differencing Regression Testing Technique.” In: *Proceedings of the International Conference on Software Maintenance*. ICSM ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 44–.

[38] Lee White et al. “Extended firewall for regression testing: an experience report.” In: *Journal of Software Maintenance and Evolution: Research and Practice* 20.6 (2008), pp. 419–433.

[39] Wikipedia. *Batch Processing*. 2015. URL: `https://en.wikipedia.org/wiki/Batch_processing` (visited on 07/25/2015).

[40] Wikipedia. *Database Application*. 2015. URL: `https://en.wikipedia.org/wiki/Database_application` (visited on 07/25/2015).

[41] Alan W. Williams. “Determination of Test Configurations for Pair-Wise Interaction Coverage.” In: *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*. TestCom ’00. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 2000, pp. 59–74.

[42] Alan W. Williams and Robert L. Probert. “A practical strategy for testing pair-wise coverage of network interfaces.” In: *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*. 1996, pp. 246–254.

[43] Lei Xu et al. “A Browser Compatibility Testing Method Based on Combinatorial Testing.” In: *Proceedings of the 2003 International Conference on Web Engineering*. ICWE’03. Oviedo, Spain: Springer-Verlag, 2003, pp. 310–313.

[44] Shin Yoo and Mark Harman. “Regression Testing Minimisation, Selection and Prioritisation: A Survey.” In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

Part 2: List of Papers

Paper 1

Industrial Experiences with Automated Regression Testing of a Legacy Database Application

Authors: Erik Rogstad, Erik Arisholm, Lionel Briand, Ronny Dalberg, and Marianne Rynning

Abstract

This paper presents a practical approach and tool (DART) for functional black-box regression testing of complex legacy database applications. Such applications are important to many organizations, but are often difficult to change and consequently prone to regression faults during maintenance. They also tend to be built without particular considerations for testability and can be hard to control and observe. We have therefore devised a practical solution for functional regression testing that captures the changes in database state (due to data manipulations) during the execution of a system under test. The differences in changed database states between consecutive executions of the system under test, on different system versions, can help identify potential regression faults. In order to make the regression test approach scalable for large, complex database applications, classification tree models are used to prioritize test cases. The test case prioritization can be applied to reduce test execution costs and analysis effort. We report on how DART was applied and evaluated on business critical batch jobs in a legacy database application in an industrial setting, namely the Norwegian Tax Accounting System (SOFIE) at the Norwegian Tax Department (NTD). DART has shown promising fault detection capabilities and cost-effectiveness and has contributed to identify many critical regression faults for the past eight releases of SOFIE.

Keywords – Regression Testing, Legacy Database Applications, Industrial Context

1 Introduction

There exist many large legacy systems with a long, often unforeseeable life span as they continue to provide core business value to their organization. A commonality of these systems is that they are difficult to change and consequently prone to regression faults. They were built on old technology and usually not constructed with consideration for testability.

For example, SOFIE is a legacy system in the Norwegian Tax Department (NTD) that has been maintained for several years. As a result of extensive internal testing and a large user base over a long period of time, the core system features are reasonably dependable. However, changes will always take place due to changed taxation laws, changed user requirements, fault corrections, and refactoring. Furthermore the release cycle of the project is rather ambitious with continuous production fixes, monthly releases for less critical fixes along with overlapping releases for new features. This continuous change process combined with the growing size and complexity of the system has increased the need for systematic regression testing over and above what the current manual testing processes can handle.

Unfortunately, existing tools and large parts of the research in the area of regression test automation focus on solutions for systems that are designed to be highly testable. This motivated NTD to establish a cooperation project with Simula Research Laboratory, in order to investigate the possibilities for more cost-effective solutions for regression testing of large legacy database applications. Through this cooperation project we have developed a novel tool, that addresses the particular needs for regression testing in NTD and, we believe, those of many legacy database applications. The tool is called DART, which is an acronym for [DA]tabase [R]egression [T]esting.

The main contributions of this paper are:

- A practical approach and tool (DART) for regression testing of database applications, with a focus on generating and prioritizing black-box test cases, automatically identifying potential regression faults and then prioritizing their inspections for early fault detection.

- Application and evaluation of DART for business critical batch jobs in a legacy database application in an industrial setting.

The remainder of this paper is organized as follows. Section 2 describes the SOFIE system and what we consider to be the major testing needs of the system. Section 3 elaborates on the testing requirements and how they are related to existing work. Section 4 describes our proposed solution, the DART tool, whereas Section 5 presents practical experiences. Finally, Section 6 concludes and describes future work.

2 Testing requirements for the SOFIE system

SOFIE is the tax accounting system in Norway, handling yearly tax revenues of approximately 500 Billion NOK. It has evolved over the past 10 years to provide dependable, automated, efficient and integrated services to all 430 tax municipalities and more than 3,000 end users (e.g., taxation officers). The system is still evolving and the maintenance project currently staffs more than 100 employees and consultants.

The system was mainly designed to handle large amounts of data, which requires high throughput and a continuous focus on performance related aspects. The system has to keep historical data for all taxpayers in Norway for at least ten years, and some of the system tables currently hold more than 500,000,000 rows of data. To handle the enormous amounts of data, the system was built as a database application on the Oracle platform. To ensure efficient data processing the business logic of the system was organized into batch jobs, along with graphical user interfaces to drive the work processes of the end users. Both system components are tightly coupled with the underlying database.

SOFIE has approximately 380 batch jobs constituting 1.7 million lines of PL/SQL code. There are four categories of batch jobs:

- Interface jobs, which read and write files and transform data between SOFIE and external systems.
- Document production jobs, which produce documents to taxpayers.
- Report jobs, which produce reports for end users.
- Core business jobs, which carry out the core business logic of the system and drives the work processes of the end users.

The batch jobs are continuously changing, and in general they are very complex, hard to test, and prone to regression faults. It is vital for NTD to avoid releasing defects in the core of the system. As the system serves all taxpayers in Norway, even "minor" defects can potentially harm Norwegian society and cause nationwide, bad press. Hence, one main testing requirement of SOFIE is the need for efficient, cost effective and reliable regression testing of the batch jobs in the system.

3 Problem definition and related work

In our context, a regression test solution must handle the following properties of the system under test:

- A batch job consists of a large number of tightly integrated set of operations, which makes it hard to control the job during test. A batch job can only be started, without further mechanisms of control. Then it runs to completion, typically in multiple, parallel job streams. Thus, you can control the input of the batch, and check the end result of it, but what happens in between is difficult to observe, and even more difficult to control.

- For the very same reasons it is very hard to build an automated test oracle (predicting the "expected result") for the system under test.

- Given the amount of batch jobs in the system, it is unrealistic to refactor them for improved testability. It would simply not be cost effective. Hence, they must be tested as they are.

Yoo and Harman [12] recently conducted a survey on regression testing minimization, selection and prioritization, constituting nearly 200 papers. It encompasses the main research results around regression testing, addressing the problems of identifying obsolete, reusable and re-testable test cases (selection), eliminate redundant test cases (minimization) and order test cases to maximize early fault detection (prioritization). The survey shows that the majority of the works focuses on white-box testing strategies, concerning relatively small stand-alone programs written in C or Java, or for spreadsheets, GUIs and web applications. The techniques surveyed presuppose an already existing, effective test suite on which to select, minimize and prioritize test cases for the regression test. Before addressing these issues, we needed to take one step back to figure out how we should collect a test baseline, and how to perform regression testing.

Chays et al. [4] noted the lack of uniform methods and testing tools for verifying the correct behavior of database applications, despite their crucial role in the operation of nearly all modern organizations. Most literature in the field was aimed at assessing performance of database management systems rather than testing the database application system for functional correctness, let alone regression testing. The authors proposed a framework for functional testing of database applications called AGENDA [3, 5, 6]. However, the framework was not intended for regression testing and we found some of the ideas hard to scale, which had only been evaluated for smaller examples.

The most relevant work we found targeting regression testing for database applications was the SIKOSA project [7, 2, 8]. The authors proposed a capture-and-replay tool for carrying out black-box regression testing of database applications. This aligned well with our objectives regarding database regression testing, namely a capture-and-replay approach, similar to what has been more commonly used for GUI testing, to automatically identify differences between the results of two identical test runs (referred to as *deviations*).

Because it is hard to build a precise test oracle for database applications with very complex queries, a more practical strategy is to capture a set of test case executions of the system under test, under the assumption that it currently works correctly (the *baseline*), and then use the replay run after modifications (the *delta*) to identify deviations and thus potential regression faults. Note that because such deviations only indicate *potential* faults, as they may also be due to valid changes, a technique is also needed to identify *actual* faults in a cost-effective manner.

The SIKOSA project restricted their work to checking input-output relations of database applications, as they stated that checking the state of the database after each test run was prohibitively expensive and difficult to implement for black-box regression testing. In our context, however, the outputs of the batch jobs are reflected directly in the database state and must therefore be monitored. The SIKOSA project provided some experimental performance measures for their tool, but did not refer to any evaluations regarding fault-detection effectiveness or cost-effectiveness, let alone in an industrial setting. Furthermore, neither of the proposed tools from the AGENDA framework or the SIKOSA project are publically available.

We also needed a specification-based, black-box testing technique to help specify test input data (test cases) with adequate coverage, based on an analysis of the input domain for a given batch job. There are many suitable tools for this purpose, but we found that the classification tree modeling technique and the supporting tool CTE-XL [9], which is built on the well-known category-partition approach [11], was both easy to use and scaled up to the kinds of input domains under consideration (e.g., more than 100 categories or classifications in one model).

We also investigated Oracles Real Application Testing (RAT) [10], but found that it was mainly targeted towards performance testing and not easily adaptable for functional testing.

In summary, the research literature provided us with a useful starting point, but none of the related works fully and directly addressed our needs, and except for CTE-XL, we could find no accessible tools to apply directly into our project context.

4 DART

The above discussions motivated the development of the DART tool, which is a tool for regression testing of database applications, and mainly targeted towards database intensive batch jobs.

The basic principle of the tool is straightforward: Execute the system under test twice on the exact same input data and initial database state, once with the original version of

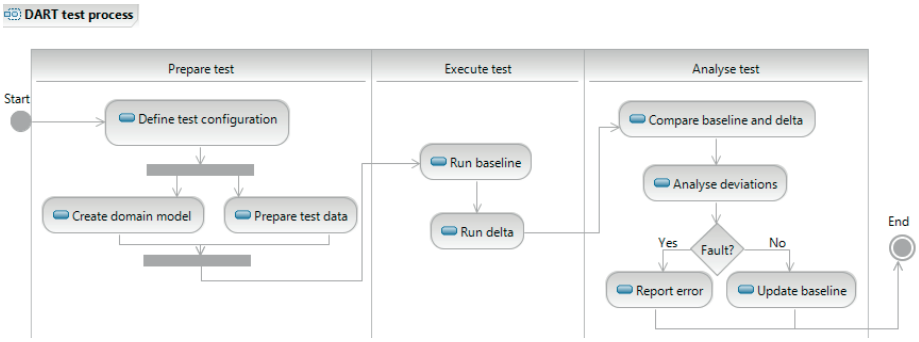


Figure 1.1: A UML Activity Diagram of the DART regression test process.

the system (baseline), and once with the changed version of the system (delta). Compute the difference in database state between the two runs. A difference is either due to a valid change, or a regression fault.

Note that DART can be used to identify regression faults in any system or program unit performing Create, Read, Update and Delete (CRUD) operations on a database, and is not restricted to batch testing only. But in our context the system under test consist of batch jobs that perform complex CRUD operations on a database, guided by business logic that implements sequences of the taxation laws and rules. There are two properties of these batch jobs that make the DART approach suitable:

- Batch jobs are built to run to completion without any manual intervention. This eases the test execution and ensures consistency between the baseline and delta run of the test.
- Batch jobs operate on a limited set of database entities. This simplifies the test setup, as the tables to monitor can be easily identified prior to the test execution.

Figure 1.1 shows the main steps in the testing process with DART. In the following sections, these steps will be described in detail.

4.1 Running example

Throughout the description of DART, a running example will be used to demonstrate the various steps of the test process. The example is intentionally kept very simple to fit size constraints. The system under test used as example is the program P shown in Figure 1.2. We use a Java-like syntax augmented with directly executable SQL statements in order to make it easier to understand for readers not acquainted with PL/SQL. It is a program that contains features for maintaining customer orders, more specifically adding

Program P

```
void removeItemFromOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                 where Order.orderNr = orderNr;
  if(orderId != null) {
    delete from Item where Item.itemName = itemName and
    Item.orderId = orderId;
    update Order set Order.changedDate = tomorrow
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void addItemToOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                 where Order.orderNr = orderNr;
  if(orderId != null)
    insert into Item (ItemName, OrderId) values ( itemName
orderId);
    update Order set Order.changedDate = tomorrow
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void main() {
  addItemToOrder(12345, "USB stick");
  removeItemFromOrder(34567, "Mouse");
}
```

Figure 1.2: The example program P.

and deleting items from a customer order. As an example execution of the program, one item is added to a customer order, while an item is removed from another customer order in the main method.

The relational entity model of the example program is shown in Figure 1.3, along with the initial state of the database prior to test execution. It consists of three entities containing information about customers and their orders. A customer can have zero to many orders with zero to many items.

4.2 Test configuration

A test with DART is set up by selecting the database tables and more specifically the table columns to monitor during the test execution. DART obtains and presents the database schema(s) of the system under test and a test engineer selects the ones to monitor during the test execution. In our example the tester would be presented with the three tables Customer, Order and Item, which all are a part of the database schema for program P. Since the program P performs operations on the two tables Order and Item, these are the ones that make sense to monitor while testing P. The tester selects the two tables and

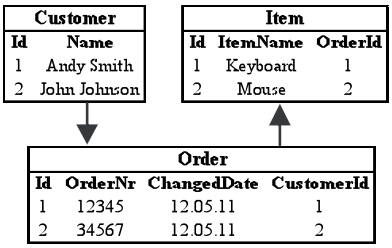


Figure 1.3: The relational entity model and initial state for program P.

more specifically the underlying table columns to monitor.

Additionally the test engineer specifies how CRUD-operations on the selected entities should be grouped together as "logical test cases" based on a meaningful, common test case identifier, e.g., a social security number. Such identifier is defined using table attributes such as primary keys, foreign keys and/or SQL queries. The goal is to logically group related rows in the tables monitored in a test execution to facilitate the comparison between the baseline and delta test executions. A meaningful common test case identifier in our example would be the customer name (assumed to be unique), as all orders and items can be traced back to its customer. In that case one customer will make out one test case and all data manipulations that are logged during test execution will be grouped by customer name. A test configuration for program P would then look like the one shown in Table 1.1.

Table 1.1: Test configuration for program P.

Table	Table column	Test case identifier
Order	OrderNr, ChangedDate	Customer.Name
Item	ItemName	Order.Customer.Name

It is also possible to give aliases to the tables and table columns in the test configuration as some tables might come from external parties and have non-intuitive names. The aliases defined in the test configuration will later on be used in the presentation of the test results. In summary a test configuration denotes the set of table columns (and their aliases) to monitor during test execution and the corresponding specification of the test case grouping scheme.

4.3 Domain modeling

Prior to test execution, test data have to be prepared for the specific system component to be tested. Whether the test data is real system data, or generated synthetically, the

output of the test data preparation process is a test suite on which the system under test can be executed. A test suite can potentially contain a large number of test cases, and there may not be enough resources available to execute all of them, or to analyze all the resulting deviations during regression testing. In particular, test suites based on real system data tend to contain large amounts of redundant test cases (as will be elaborated in Section 5.2), which will result in duplicate deviations causing unnecessary inspections. Hence, to alleviate this problem, we would like to prioritize the test cases in a test suite to ensure that we execute first test cases that are most likely to reveal distinct regression faults.

In order to prioritize the test cases in the test suite, a model of the domain under test is made using the tool CTE-XL. The model is a classification tree (defining equivalence classes), which is used to generate domain partitions (also called test case specifications or abstract test cases) according to a coverage criterion of your choice, for example pairwise coverage of the equivalence classes. A domain model for the example program P can look like the one shown in Figure 1.4. The root node Program P, the classifications Number of orders for customer, Item added and Item deleted, and the classes (0, 1, >1) and (Y, N) constitute the classification tree model, whereas the bottom six lines each represent partitions. In this case the pairwise coverage criterion was used to generate the partitions, which ensures that each pair of classes are represented in at least one partition. The test model emphasize the following aspects regarding the program P:

- The number of orders for a particular customer matters. If a customer has zero orders an error should be reported, otherwise the item should be added or deleted. It is also interesting to differentiate the case of a customer having more than one order, to make sure items are removed and added to the right order and only that.

- It is also interesting to test different variations of adding and deleting items for different numbers of customer orders.

Given a test suite and a domain model of the system, DART provides the capability of matching the data in the test suite with the partitions in the domain model. In the initial state of program P shown in Figure 1.3, there are two test cases, namely customer Andy Smith and John Johnson. DART will match the test case Andy Smith with Partition 3 as he has one order in which an item will be added, and the test case John Johnson with Partition 4 as he has one order in which an item will be deleted. When the test cases have been matched to partitions, DART prioritizes the test cases as follows: Among the partitions containing test cases, select a random partition, and a random test case within the partition. Next, select again a random partition among the remaining ones, and again a random test case within the partition. Continue until all partitions have been

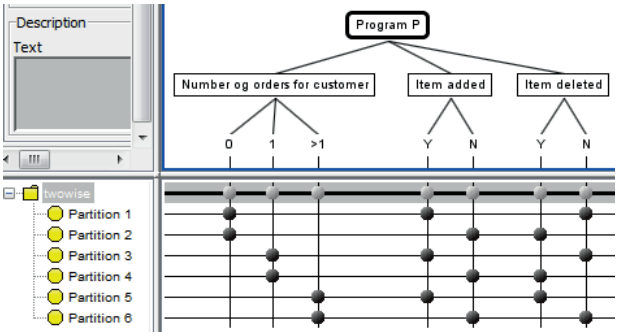


Figure 1.4: A classification tree model for program P.

selected. Then start the process again and select test cases among the ones that have not been selected yet. Stop the process when all test cases have been selected. The resulting ordering of the test case selection determines the priority of the test cases. The rationale is to ensure that all partitions be covered as quickly as possible during test execution and that, for cases where there is a deviation, the inspection of such deviations are more likely to uncover dissimilar regression faults as quickly as possible. This strategy should be considered as a first step to be improved upon, as further described in the conclusion.

In the trivial running example the prioritization is meaningless as there are only two test cases from two different partitions. However, this is important in realistic database applications, as test cases can be numerous, expensive to run, and manual inspections of deviations are time-consuming, as reported in Section 5.1. We refer to this process as a partition-based approach for test case prioritization.

4.4 Test execution

During test execution DART will log all data manipulations related to the specific test configuration. The way data manipulations are recorded and logged is through dynamically generated database triggers on the tables specified in the test configuration. A trigger is procedural code that is automatically executed in response to certain events on a table or view in a database. Pseudo-code for generating the triggers is shown in Figure 1.5. As the algorithm shows, a trigger is generated for each table in the test configuration. Each of the generated table triggers is defined to insert a row into the DART log table for each data manipulation on the columns specified in the test configuration for the given table. Insert and delete operations are always done at the row level and DART will log values for all table columns in the test configuration when an insert or delete operation takes place. Update operations can be attribute specific, so DART will only

```
Input:    T is the set of tables in the configuration
          C is the set of columns from all tables in the
configuration
          T_C: 2^(T×C) is the set of (table, column) pairs
          T_R is a test run

Algorithm generateTriggers(T_C, T_R)
begin
1. for each table t ∈ T do
2.   testCaseID ← getTestCaseID(t, T_R, oldValue, newValue);
3.   triggerStatement ← "Create trigger on table t that fires after
                        insert, delete or update ";

4.   for each (t,c) ∈ T_C do
5.     triggerStatement.append("if insert operation then insert
           (testCaseNr, t, c, inserted value) into DART log table");
6.     triggerStatement.append("if update operation then
           if update on column c then insert(testCaseID,
           t, c, old value, new value) into DART log table");
7.     triggerStatement.append( "if delete operation then insert
           (testCaseID, t, c, deleted value) into DART log table");
     end loop;
   end loop;
8. execute triggerStatement;
end
```

Figure 1.5: Algorithm for trigger generation in DART.

log the table columns in the test configuration that is actually updated. The triggers are dynamically generated as a Data Definition Language (DDL) string, which is executed in the end to store the actual triggers in the database.

Thus, DART dynamically instruments the database of the system under test by generating test-configuration specific database triggers when the test is started. During test execution these triggers will fire on any insert, delete or update on the table columns in the test configuration and store the database operations into a DART log table. One data manipulation operation results into one row in the log table matching the format <test case identifier, table name, column name, old value, new value>. The test case identifier (e.g., the customer name) is what uniquely identifies the test case that causes the operation to be executed. It is devised on the fly according to the specification in the test configuration. Table name and column name are the names of the table and column the operation is executed on, respectively. Old value and new value refer to the values of the attribute prior to and after the operation execution, respectively. Old value is given the static value "Inserted" for insert operations, while new value is given the static value "Deleted" for delete operations. After test execution the triggers are deleted from the database of the system under test.

A test run is done once with the original version of the system (baseline) and once with the changed version of the system (delta), which is subject to regression faults. Before the delta test run the database is reset to the initial (baseline) state to ensure that

both runs start out with the same database state. Various mechanisms are available to reset the database. We have used the flashback to restore point feature of Oracle in the particular case of SOFIE. This is done by creating a restore point in the database after the test configuration is defined and the test data is prepared, but before the execution of the baseline run starts. The restore point defines the state of the database at the time it is created and will ensure consistency between the test runs. In our example, a test run on program P, with the test configuration from Table 1.1 and the initial state from Figure 1.3, would result in the DART log data shown in Table 1.2. For the test case Andy Smith, one insert operation and one update operation is executed, as logged in row 1 and 2 of Table 1.2, respectively. For the test case John Johnson one delete operation and one update operation is executed, as logged in row 3 and 4 of Table 1.2, respectively.

Table 1.2: Example DART log table after the baseline run.

Id	Test run id	Test case	Table name	Column name	Old value	New value
1	1	Andy Smith	Item	Item Name	Inserted	USB Stick
2	1	Andy Smith	Order	Changed Date	12.05.11	14.05.11
3	1	John Johnson	Item	Item Name	Keyboard	Deleted
4	1	John Johnson	Order	Changed Date	12.05.11	14.05.11

It turns out that program P contains a fault. The changedDate of the order should be updated to today's date when an order is changed. Currently it is updated to tomorrow's date. The fault is corrected (underlined) and a new version of P, called P' is shown in Figure 1.6. For illustration purposes let us assume a regression fault in P': the update of the order in removeItemFromOrder method is completely removed, rather than fixed (line struck through). After resetting the database into the same initial state as before the first test run, the test is executed again on the changed program version P'.

After both test runs, the DART log table contains the information shown in Table 1.3. Three additional rows are logged for the delta run. An insert and an update operation for the test case Andy Smith in row 5 and 6, and a delete operation for the test case John Johnson in row 7.

Table 1.3: Example DART log table after both test runs are executed.

Id	Test run id	Test case	Table name	Column name	Old value	New value
1	1	Andy Smith	Item	Item Name	Inserted	USB Stick
2	1	Andy Smith	Order	Changed Date	12.05.11	14.05.11
3	1	John Johnson	Item	Item Name	Keyboard	Deleted
4	1	John Johnson	Order	Changed Date	12.05.11	14.05.11
5	2	Andy Smith	Item	Item Name	Inserted	USB Stick
6	2	Andy Smith	Order	Changed Date	12.05.11	13.05.11
7	2	John Johnson	Item	Item Name	Keyboard	Deleted

Program P'

```
void removeItemFromOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                    where Order.orderNr = orderNr;
  if(orderId != null) {
    delete from Item where Item.itemName = itemName and
    Item.orderId = orderId;
    update Order set Order.changedDate = tomorrow
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void addItemToOrder(Long orderNr, String itemName) {
  Long orderId = select Id from Order
                    where Order.orderNr = orderNr;
  if(orderId != null)
    insert into Item (ItemName, OrderId) values ( itemName
orderId);
    update Order set Order.changedDate = today
    where Order.Id = orderId;
  } else{
    reportOrderDoNotExistError();
  }
}

void main() {
  addItemToOrder(12345, "USB stick");
  removeItemFromOrder(34567, "Mouse");
}
```

Figure 1.6: The example program P', which is a modified version of program P.

4.5 Test Analysis

After a test is executed on two different versions of the system under test, the two test runs are compared with each other. The output of the test execution is a DART log table filled with all data manipulation operations of the respective test runs. The comparison uses the SQL set operations minus and union to compute the difference between the two runs, as follows:

<Log data from baseline> MINUS <Log data from delta>
UNION ALL
<Log data from delta> MINUS <Log data from baseline>

The comparison operation will reveal all differences between the baseline and delta runs with regards to the test configuration. The deviations, grouped by the test case identifier, are presented to the tester, which in turn has to determine whether the deviation is a regression fault or not.

In our example the output of the test is the deviations between the two runs as shown in

Table 3.1. There is one deviation due to the changed update in addItemToOrder (row 1-2) and one deviation due to the missing update in the delta version of removeItemFromOrder (row 3). By analyzing the deviations in Table 3.1, the test engineer can verify that the change in test case Andy Smith is due to correct changes in P', whereas the missing update in the test case John Johnson is due to a regression fault.

As the baseline run essentially serves as the test oracle, DART will identify regression faults introduced in the delta version of the system, but will not identify faults that are present in both the baseline and delta run. In practice, the same baseline is used for testing several consecutive deltas. After each test, the deviations that are correct in the delta are updated into the baseline. Thus, the baseline is continually improved and the test oracle increasingly more accurate.

Table 1.4: The deviations between the test runs for P and P'.

Id	Test case	Table name	Column name	Old value	New value	Test run
1	Andy Smith	Order	Changed Date	12.05.11	14.05.11	Baseline
2	Andy Smith	Order	Changed Date	12.05.11	13.05.11	Delta
3	John Johnson	Order	Changed Date	12.05.11	14.05.11	Baseline

5 Practical Experiences

5.1 Pilot evaluation

During the development of DART we conducted a pilot evaluation of the tool to investigate its regression fault detection capabilities. In our pilot study we chose to focus on one particular functional area of the system, the most complex and business critical one. Due to its complexity this is an area that has been prone to regression faults in the past. Since all taxpayers in Norway could be affected, it is of great importance to avoid faults. This particular functional area consists of 19 different batch jobs.

For the pilot we chose to test a previous system release, which had already undergone the regular, manual testing and QA activities. One part of the selected functional domain had been refactored in that release. As a result, five regression faults had been identified during the regular testing routines in the project. Additionally five regression faults had been discovered in the production environment after it was released. As a pilot evaluation, we were interested to see if we could identify the same ten regression faults, and possibly additional, undiscovered faults, with the DART tool. We compared the last version of the system prior to the refactoring with the version that was delivered to the system test in that particular release.

For the pilot we had three sets of real system test data available. The test suites were of different sizes and for evaluation purposes we chose to run the regression test for all three of them. The test data in the three test suites consisted of non-overlapping test cases, where each test case represented one taxpayer. Table 1.5 summarizes the three test runs. Column two shows the number of test cases contained in the test suites, column three shows how many of the test cases deviated between the baseline and delta run, column four shows how many of the deviations were due to valid changes, column five shows how many of the deviations were due to regression faults, column six shows the number of distinct functional faults among the faulty deviations, column seven shows the number of faults that had been detected during testing and operation, which were rediscovered with DART, column eight shows the new regression faults detected by DART and column nine shows the inspection effort spent determining whether the deviations were correct or faulty.

Table 1.5: Summary of test runs in the pilot evaluation.

Test	# **Test cases**	# **Deviations**	# **Correct deviations**	# **Faulty deviations**	# **Distinct faults**	# **Previous faults found**	# **New faults found**	**Inspection time**
1	711	33	19	14	7	5	2	7 hours
2	3144	182	136	46	11	7	4	35 hours
3	5670	522	386	136	15	6	9	105 hours

DART revealed eight of the ten faults that were previously found during testing and operation, but also helped identify nine undiscovered faults, that is, nine faults that were still present in the production system and needed to be corrected. In total, the three test runs uncovered 17 distinct faults. The two previously detected faults missed by DART were not found due to the insufficient coverage of the test suites; none of the test cases in the three test suites exercised the two faulty situations.

As expected, the largest number of faults was found in the largest test suite, but its set of detected faults did not subsume those of the smaller test suites; two of the faults discovered in the smaller test suites were not present in the largest one. This suggested that we needed a more systematic way to specify the regression test cases, as elaborated in the next section (5.2). Nevertheless, as a result of the pilot we registered nine new defects in the defect tracking system. One of them was registered as a “A defect”, seven as “B defects” and one as a “C defect” on a criticality scale ranging from A to C, where A is the most critical one. Broadly speaking, A defects are critical, B defects are serious, while C defects are less important.

For the purpose of the evaluation, we analyzed all deviations in the three test runs to ensure that we found as many defects as possible. However, this required a considerable amount of manual effort, as shown in Table 1.5 (Inspection time); on average we used

about 12 minutes per deviation. This suggests that, in order to use DART for large-scale regression testing in a system release, we would need a way to prioritize test cases to increase the likelihood of early fault detection and reduce the number of redundant deviations to analyze. The same functional fault was present in several deviations, and ideally we would only like to inspect one deviation for each unique functional fault. Thus, a classification tree model of the input domain was developed and applied to prioritize test cases, as described in Section 4.3.

We applied the prioritization to the test cases in test suite 3 as it was the largest. Figure 1.7 shows the results of using the partition-based approach for prioritizing test cases to execute n test cases and analyze the resulting deviations in their given priority order for various values of n. The results are then compared to the average resulting from the random selection of test cases. To obtain the results in Figure 7, we repeated the prioritization procedure 100 times and averaged the percentage of faults detected (the Y-axis) for a given percentage of test cases in the test suite (the X-axis). Though the results are very clear just by looking at Figure 7, to check the statistical significance of the difference between the partition-based approach over the random approach, we conducted non-parametric Mann-Whitney U-Tests [1] to test the difference in fault detection for each test suite size value. We computed p-values for all sizes that were sampled and all of them were below $\alpha = 0.05$, showing that the two approaches are significantly different. More precisely the p-value was less than 0.0000002 from 1 to 90 percent of the test cases, and 0.01381 for 95 percent. We tested the entire set of sample data from the two approaches, which yielded a p-value of 0.00019.

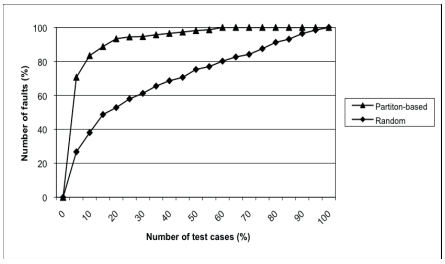


Figure 1.7: A comparison of partition-based-, and random test case selection.

In practice this means that for example by only executing eight percent of the test cases and analyze the resulting deviations, the test engineer would on average find more than 80 percent of the faults. This corresponds to executing approximately 450 test cases, which on average resulted in 80 deviations uncovering 12 out of 15 faults that could be uncovered by the test suite. In terms of effort that is 16 hours of inspection time for revealing 12 out of 15 faults. In comparison, with a random selection strategy, we would

on average have found less than 35 percent of the faults for a similarly sized test suite. We consider this to be a substantial, practically important cost saving.

To summarize, the pilot evaluation showed that DART could help detect significantly more regression faults and that the test case prioritization using DART could yield significant savings in terms of number of test case executions and the effort involved in analyzing deviations.

5.2 Test coverage and synthetic test data

As mentioned in the previous section, the test suites in the pilot evaluation uncovered a total of 17 faults. These test suites were based on live data input files provided by the operation environment. It turned out that none of the three test suites were, in isolation, adequate to reveal all the 17 faults. Neither did they uncover all the ten faults previously identified during test and operation, indicating that not even combining the three test suites yields satisfactory coverage. Considering the complexity of the domain model for the system under test, this is not surprising when the test data were not derived in a systematic manner.

By applying the all combination coverage criterion on the domain model for that particular functional area, as many as 17,100 partitions were generated. To assess how well live test data would cover those partitions, we selected a large, representative test suite consisting of 211,837 "live" test cases (actual tax payers), provided by the production environment, and compared it with the partitions. We found that the test suite covered only 226 out of 17,100 partitions, a model coverage of only 1.32 percent! The two largest partitions of the test suite contained 86,743 and 36,296 test cases, respectively, showing huge numbers of redundant test cases while showing serious shortcomings in covering exceptional cases (rare patterns of taxpayers). Live test data also entail practical concerns. Confidentiality issues must be addressed. They are not always available, as one may depend on third parties to deliver them and they are hard to reuse, as they are dependent on a given database state.

The lack of model coverage achieved with live test data along with their associated practical concerns motivated the generation of synthetic test data. To drive the generation of synthetic test data, we use the same domain model as we use for partition-based test case prioritization. Adapter code is written to map the abstract values of the leaf classes in the classification tree model to actual parameter values of the real test cases for the system under test. Using the adapter code, test cases can be automatically generated according to the model. This makes it easy to generate different test suites, providing different levels of model coverage, e.g., two-wise, three-wise, or all combinations. The usage of synthetically generated test data with DART is still in its initial phase. A few

system faults were identified while developing the adapter code, as rare system scenarios got executed. We are confident that the generation of synthetic test data will allow us to increase test coverage and make testing more efficient and predictable when applied in DART.

5.3 Deployment into project setting

DART has been used to support regression testing of batch jobs in the core functional areas of the SOFIE application for the past eight releases. So far we have used DART as a supplement to manual testing, not as a replacement. We thus had the opportunity to compare the fault detection effectiveness of DART with the regular (manual) system testing routines in the project. Table 1.6 shows the faults detected in the eight releases during regular system testing and the additional faults detected by DART, within the particular functional area of interest. It also shows the number of faults that slipped through both testing activities, but were later on detected during operation in the production environment.

Table 1.6: Defects detected in the past eight releases of SOFIE.

Releases	**# Faults detected by regular testing**	**# Additional faults detected by DART**	**# Faults discovered in production**
1	6	9	6
2	3	1	1
3	1	1	1
4	6	2	3
5	0	0	0
6	19	3	2
7	1	5	1
8	1	1	0
Total	37	22	14

The figures in Table 1.6 are meant to give a rough picture of the impact of DART during its initial lifetime in the SOFIE project. Unfortunately, we do not have exact information about the effort spent for uncovering the faults by the different testing approaches, as we faced organizational challenges in the project while trying to get the time reported at a satisfactory level for evaluation. However, the faults uncovered by regular testing are typically the result of weeks of testing, while the faults uncovered by DART result from days of testing. It is also worth stressing that we had no regression test environment in place in the first six releases shown in Table 1.6. Consequently, DART was not used during the test period, but rather as a final verification of the releases after the acceptance

test was finished and the release was ready to ship. Therefore, the figures provided in Table 1.6 should not be used to strictly compare the fault detection capabilities of DART with those of the regular testing routines, as DART could only detect the leftover faults in the first six releases. Table 1.6 shows that DART has helped uncover more than a third of the defects found during regression testing (22 out of 59), within the batches of the core functional domain. Put in other words DART has helped identify approximately 60% more regression faults than what would have been detected without it. We consider this to be of substantial impact, especially since DART was only used as a "last check" in the first six releases. Such results combined with the savings discussed in Section 5.1, make us confident that the test team can now rely on DART for regression testing of the batch jobs in SOFIE, while reassigning some of their resources on other types of testing. For example, faults in the graphical user interfaces, documents and reports within the same functional domain were discovered by regular testing routines, but would not have been found by DART. The same applies to the extensive testing required to verify the correctness of new functionality. An example of the latter is release six in Table 1.6, where substantial new functionality was introduced, and thoroughly tested, revealing several faults in the regular testing routines.

Even when combining manual testing with DART, some faults still slipped through into production, as shown in Table 1.6. As an evaluation of the DART tool, we went through the defects reported from the production environment to understand why they were not discovered prior to being released. Table 1.7 lists the findings.

Table 1.7: Reasons why defects reported from production were not found by DART.

# Defects	Cause of not being detected by DART prior to release
3	Insufficient test partition coverage to reveal the fault; no test cases that executed the faulty situations.
1	Did not execute that part of the functional domain in that particular test.
2	Found and reported by DART, but there were not enough time to fix them prior to release. Also reported from the production environment before they got fixed.
2	Performance issue specific to the production environment.
4	Found and reported by DART, but the test was executed after the release (pilot evaluation).
2	Currently unknown due to lacking information regarding the faults.

Six of the defects were actually discovered by DART. One was not found as we ran the test on a limited scope in the beginning, before broadening our horizon the whole batch process of the functional domain in the later releases. Two of the faults were performance-related issues only present in the production environment (due to different

settings). Besides the two currently unknown defects, that leaves us with three defects that should have been detected, but were not due to insufficient partition coverage. We hope to address this issue in the future by synthetically generating test data, as discussed in the previous section.

For the sake of the evaluation we also investigated the criticality distribution of the defects reported from manual testing, DART, and production. No conclusion could be drawn regarding the relationship between the criticality of defects and how they were detected.

Another important contribution of DART in practice is that it has impacted the prioritization of defects in the project. Since DART enables more thorough and cost-effective regression testing, less defect corrections are postponed due to their high risk of generating regression faults. In practice that means that more faults are corrected more quickly, while still remaining confident that they do not introduce new regression faults.

6 Conclusion and future work

We have reported our experience with a practical approach and tool (DART) for functional black-box regression testing of legacy database applications. The tool uses dynamically generated database triggers to capture the data manipulations in the database during execution of the system under test. The difference between consecutive executions on different versions of the system under test is used to identify regression faults. The tool makes use of CTE-XL classification tree models to prioritize test cases and minimize their redundancy, so as to make our approach scalable to real system releases. The prioritization mechanism increases the likelihood of early fault detection and can be used to both reduce execution time and the effort involved in analyzing differences.

In this paper, our approach was applied on batch jobs in the Norwegian Tax Accounting System SOFIE, a very large database application. However, we believe our results are applicable outside this context, and for any program performing CRUD operations on a database. DART has shown good fault detection capabilities on multiple SOFIE releases. In the pilot evaluation, where DART was applied to a system release that had already been tested and released, DART found eight of the ten regression faults that were uncovered during regular testing and system operation, but also detected nine additional regression faults. For the past eight releases of SOFIE, DART has been used as a support tool for regression testing, and has helped identified 60% additional faults, that would have been released otherwise. Thanks to DART, the business critical batch jobs in SOFIE are more thoroughly, yet efficiently tested, causing less regression faults to be released. This enables NTD to take more risks by correcting more bugs in shorter periods of time.

Current work in progress is to fully integrate DART with the daily test operation of the project, and ideally as a continuous part of the development process, as a means for early fault detection. We will continue to work on generation of synthetic test data and use them for test execution with DART to ensure better test coverage and more efficient and predictable testing.

We have applied a relatively simple, yet efficient method for test case prioritization. More work is required to determine the optimal way for test case prioritization based on a classification tree model. For example, similarity measurement between partitions and test cases could be used to refine the prioritization of test cases.

Finally our ambition is to replace the current Oracle specific version of DART with a fully implemented open source Java version, to address the lack of good tool support for regression testing of database applications.

7 Acknowledgement

We are grateful to Hilde Lyngstad, Trond Andreassen, Thor-Otto Thuresson and Bjørn-Erik Godøy for their contributions in the project.

8 References

[1] Andrea Arcuri and Lionel Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In: *Software Engineering (ICSE), 2011 33rd International Conference on*. 2011, pp. 1–10.

[2] Carsten Binnig, Donald Kossmann, and Eric Lo. "Testing Database Applications." In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 739–741.

[3] David Chays and Yuetang Deng. "Demonstration of AGENDA tool set for testing relational database applications." In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. 2003, pp. 802–803.

[4] David Chays et al. "A Framework for Testing Database Applications." In: *SIGSOFT Softw. Eng. Notes* 25.5 (Aug. 2000), pp. 147–157.

[5] David Chays et al. "An AGENDA for Testing Relational Database Applications: Research Articles." In: *Software Testing, Verification and Reliability* 14.1 (Mar. 2004), pp. 17–44.

[6] Yuetang Deng, Phyllis Frankl, and David Chays. “Testing database transactions with AGENDA.” In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. 2005, pp. 78–87.

[7] Florian Haftmann, Donald Kossmann, and Er Kreutz. “Efficient regression tests for database applications.” In: *In Conference on Innovative Data Systems Research (CIDR*. 2005, pp. 95–106.

[8] Florian Haftmann, Donald Kossmann, and Eric Lo. “A Framework for Efficient Regression Tests on Database Applications.” In: *The VLDB Journal* 16.1 (Jan. 2007), pp. 145–164.

[9] Berner & Matter. *CTE-XL and CTE-XL Professional - Overview*. May 2011. URL: `http://www.berner-mattner.com/en/berner-mattner-home/products/cte-xl/`.

[10] Oracle. *Oracle Real Application Testing*. May 2011. URL: `http://www.oracle.com/us/products/database/options/real-application-testing/overview/index.html`.

[11] Thomas Joseph Ostrand and Marc J. Balcer. “The category-partition method for specifying and generating fuctional tests.” In: *Communications of the ACM* 31.6 (1988), pp. 676–686.

[12] Shin Yoo and Mark Harman. “Regression Testing Minimisation, Selection and Prioritisation: A Survey.” In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

Paper 2

Test Case Selection for Black-Box Regression Testing of Database Applications

Authors: Erik Rogstad, Lionel Briand, and Richard Torkar

Abstract

Context: This paper presents an approach for selecting regression test cases in the context of large-scale database applications. We focus on a black-box (specification-based) approach, relying on classification tree models to model the input domain of the system under test (SUT), in order to obtain a more practical and scalable solution. We perform an experiment in an industrial setting where the SUT is a large database application in Norway's tax department.

Objective: We investigate the use of similarity-based test case selection for supporting black box regression testing of database applications. We have developed a practical approach and tool (DART) for functional black-box regression testing of database applications. In order to make the regression test approach scalable for large database applications, we needed a test case selection strategy that reduces the test execution costs and analysis effort. We used classification tree models to partition the input domain of the SUT in order to then select test cases. Rather than selecting test cases at random from each partition, we incorporated a similarity-based test case selection, hypothesizing that it would yield a higher fault detection rate.

Method: An experiment was conducted to determine which similarity-based selection algorithm was the most suitable in selecting test cases in large regression test suites, and whether similarity-based selection was a worthwhile and practical alternative to simpler solutions.

Results: The results show that combining similarity measurement with partition-based test case selection, by using similarity-based test case selection within each partition, can provide improved fault detection rates over simpler solutions when specific conditions are met regarding the partitions.

Conclusions: Under the conditions present in the experiment the improvements were marginal. However, a detailed analysis concludes that the similarity-based selection strategy should be applied when a large number of test cases are contained in each partition and there is significant variability within partitions. If these conditions are not present, incorporating similarity measures is not worthwhile, since the gain is negligible over a random selection within each partition.

Keywords – Test case selection, Regression testing, Database applications, Similarity measures

1 Introduction

Regression testing is known to be a very expensive activity as, in most cases, regression test suites are large since they attempt to exercise the system under test in a comprehensive manner. This problem is more particularly acute when test results must be manually checked, running test cases is expensive, or when access to a test infrastructure is required. As a result, as reported in a very abundant research literature [20], regression test cases must be carefully selected or prioritized. Most selection strategies are based on the static analysis of source code structure and changes. However, in many situations, this is not practical (lack of proper tool support) or applicable (no direct access to the source code or third party components) and a black-box approach, based on the system specifications, must be adopted.

In this paper, we investigate strategies for selecting regression test cases based on classification tree models. Such models have been traditionally used to partition the input domain of the system being tested [13], which in turn is used to select and generate system test cases so as to achieve certain strategies for partition coverage. Such models are widely applied for black-box system testing for database applications and is therefore a natural and practical choice in our context. In other words, the goal is to minimize regression testing effort while retaining maximum fault detection power, and do so by relying on a specific model of the input domain.

More specifically, we combine similarity-based test case selection with such a partition-based approach to refine regression test selection. Similarity-based test case selection is a strategy that has shown itself to be cost-effective in other contexts [9, 8]. At a high level, our strategy consists in selecting regression test cases within partitions in order to maximize their diversity. A large scale experiment was conducted in the context of an industrial database application. The main goal was to investigate the impact of our proposed selection strategy in terms of fault detection rates, and the conditions under which it is beneficial. The contribution of this paper lies in defining a practical strategy for applying similarity measurements when selecting test cases generated from classification tree models, and in evaluating the approach to support regression testing in an industrial setting. This setting, further described below, is a large and business critical database application developed by the tax department of Norway.

Given the results presented in this paper, we recommend the use of similarity partition-based test case selection when the classification tree model is defined in such a way as to contain significant variability within the partitions, and the partitions generally contain many test cases. Under these conditions, it can be highly beneficial to use similarity measures as a means of selecting test cases within each partition of the classification tree model, as it leads to significant increases in fault detection rates.

The remainder of the paper is organized as follows: Section 2 describes the industrial context and the challenge addressed in the study. Section 3 provides background information regarding similarity-based test case selection and elaborates on similarity functions applicable for our context. Our proposed solution for selecting test cases generated from classification tree models, by incorporating similarity measurements, is presented in Section 4, whereas the results of the experiment are shown in Section 5, along with a discussion of the implications of the outcomes. Conclusions are provided in Section 6.

2 Industrial setting and problem formulation

SOFIE is the tax accounting system of Norway, handling yearly tax revenues of approximately $90 billion. It has evolved over the past 10 years to provide dependable, automated, efficient and integrated services to all 430 tax municipalities and more than 3,000 end users (e.g., taxation officers). SOFIE is still evolving to accommodate changes in the tax laws, and its maintenance currently involves more than 50 employees.

The system was mainly designed to handle large amounts of data, thus requiring high throughput and a continuous focus on performance-related aspects. Historical data for all taxpayers in Norway has to be kept for at least ten years, and some of the system tables currently hold more than 500 million rows of data. To ensure efficient data processing, the business logic of the system was organized into batch jobs, along with graphical user interfaces to drive the work-processes of the end users. Both system components are tightly coupled with the API of the underlying database management system. SOFIE has approximately 380 batch jobs constituting 1.7 million lines of PL/SQL code. The batch jobs are continuously changing during maintenance and in general they are very complex, hard to test, and prone to regression faults. Since the system serves all taxpayers in Norway, it is vital for the Norwegian Tax Department to avoid releasing defects in the core of the system.

In [18], we proposed an approach and tool (DART) for the regression testing of large database applications like SOFIE, with an emphasis on proving efficient mechanisms for testing oracles, which is the comparison of test results across releases. We presented an evaluation of DART in eight consecutive releases of SOFIE, showing promising fault detection capabilities. In this paper, we will build on this work and devise more sophisticated regression test selection strategies based on black-box models of the input domain.

In a system like SOFIE, vast amounts of data are available from the production environment. Consequently, the testing of SOFIE relies heavily on the use of real input data, leading to a test strategy based on usage profiles. Generating and maintaining such large amounts of synthetic test data would be a tedious and expensive process and it is there-

fore beneficial to rely on actual tax payer data for regression testing as well. In practice the test data is made available by making a copy of the production database for testing purposes and then reusing input files from the production environment. Thus, the set of test data will vary over time depending on the operations in the production environment. Since we do not rely on exactly the same set of test data over time, we need a cost-effective and automated strategy for selecting a subset of it that will maximize the likelihood of detecting faults. Several motivating factors exist for doing so. For one thing, the test environments are less sophisticated than the production environment regarding computer resources, thus leading to limitations in terms of the number of test cases that can be run during regression testing. Second, the set of production test data available is likely to contain large numbers of similar test cases, which are thus likely to reveal the same faults. Third, DART produces a set of deviations as regression testing outputs. Testers need to inspect them in order to identify potential regression faults as it is impossible to automatically decide whether such a deviation should be expected given changes made in the last system version. Since such inspections are costly, we would like to reduce the number of deviations to inspect in the regression test. By selecting test cases in an effective manner, the number of deviations resulting from the same fault would hopefully be limited.

Our priority in this paper is therefore to devise and evaluate cost-effective strategies to select regression test cases. Our approach to selection is black-box (specification-based), which is based on a model of the input domain. Choosing a black-box approach for a regression testing of the batch jobs in SOFIE was primarily motivated by the fact that the testers in the project have limited technical expertise regarding the system implementation, and prefer to verify system functionality based on the specifications rather than the source code. Hence, we adopted a black-box approach to regression testing that does not require source code analysis [18].

3 Background

The literature within the area of regression test case selection has primarily been focused on white-box techniques, and coverage-based selection has been the most common practice for years [20]. However, these techniques are not directly applicable within the context of black-box testing. Hemmati et al. (2010) [9] conducted a detailed study on similarity-based test case selection for model-based testing. Given a scenario in which you have to select a subset of test cases to execute, similarity-based test case selection aims to select the most diverse ones in order to increase the fault detection rate. Hemmati's approach includes three components: an encoding (representation) of test cases, a simi-

larity function, and a selection algorithm. The study showed very good results for using diversity as a means of selecting test cases. Hemmati also investigated different encoding strategies, similarity functions and selection algorithms in two industrial settings and in the context of model-based testing. Overall, results suggested that the Gower Legendre similarity function and the (1+1) Evolutionary selection algorithm was best suited for a similarity-based test case selection on test cases generated from UML state machine models [11, 10]. Cartaxo et al (2011) [8] also reported on the success of using similarity functions as a means of selecting test cases for labeled transition systems.

3.1 Similarity measures

Whereas Hemmati et al. focus on selecting a subset of test cases generated from UML state machines, we will select test cases from classification tree models for black box regression testing of database applications. Hence, it is necessary to introduce similarity measures more closely adapted to our context. As further elaborated in Section 4, the abstract test cases generated from the classification tree models are represented as series of model property types as integers, strings or Boolean values. Given the nature of these test cases, the family of geometrical similarity functions seems to be the most reasonable choice for defining similarity, since each model property could span out a dimension in the N-dimensional space. There are various such functions available, but the ones we selected for our study are *Euclidian distance*, *Manhattan distance*, *Mahalanobis distance* and *Normalized Compression Distance (NCD)*. The rationale behind this selection was to assess functions of varying complexity and assess the trade-off between complexity and selection effectiveness. Euclidian and Manhattan are simple distance measures, while Mahalanobis is a bit more sophisticated, taking the correlation between model properties into account. Despite being a different domain of application, Liparas et al. (2011) [14] recently reported good results when using Mahalanobis for defect diagnosis. NCD uses a general compression format (e.g. gzip) to group various types of objects based on their similarity. Since NCD bases its similarity measurements on a general compression format, the method is general and can be used to compare any two objects, regardless of their nature. For example, NCD has been successfully applied to cluster music files based on genres and even by composer within the specific genre of classical music [4]. A more formal presentation of each similarity measure is given below.

The Euclidian distance between two points $\vec{x}$ and $\vec{y}$ is the length of the line segment connecting them [6], given by the following formula:

$$\mathrm{Euc}(\vec{x}, \vec{y}) = \sum_{i=1}^{n} \sqrt{(y_i - x_i)^2} \tag{2.1}$$

The Manhattan distance between two points $\vec{x}$ and $\vec{y}$ is the sum of the absolute differences of their coordinates [12], given by the following formula:

$$\mathrm{Man}(\vec{x}, \vec{y}) = \sum_{i=1}^{n} |x_i - y_i| \tag{2.2}$$

The Mahalanobis distance between two points $\vec{x}$ and $\vec{y}$ is given by the following formula [15]:

$$\mathrm{Mah}(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T S^{-1} (\vec{x} - \vec{y})} \tag{2.3}$$

where S is the covariance matrix. A simplistic approach to estimating the probability that a test point in an N-dimensional space belongs to a set, is to average the center of the sample points and use the standard deviation to determine the probability. While this approach assumes a spherical distribution, Mahalanobis accounts for non-spherical distribution (i.e. ellipsoidal) by incorporating the covariance matrix. Thus it does not only take the distance from the center into account, but also the direction.

Bennett et al. have introduced a universal cognitive similarity distance called Information Distance [3]. Information Distance, which is based on Kolmogorov complexity, is, however, uncomputable. The uncomputability of the Information Distance metric can be overcome by using data compressors to approximate Kolmogorov complexity, i.e. compressors like gzip and bzip2. In [5], Cilibrasi introduced the Normalized Compression Distance, NCD, which uses compressors to approximate Kolmogorov complexity and, thus, also provides practitioners a way to indirectly use the Information Distance metric:

$$\mathrm{NCD}(x, y) = \frac{C(x, y) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}} \tag{2.4}$$

where $C(x)$ is the length of the binary string x after compression by the compressor C and $C(x, y)$ is the length of the concatenated binary string xy after compression by the compressor C. In practice, NCD is a non-negative number $0 <= r <= 1 + \epsilon$, where ϵ is small and depends on how good an approximation of Kolmogorov complexity the compressor (C) is. For modern compressors like gzip and bzip2, ϵ is typically 0.1 and more recent compressors can even come close to 0, i.e. being excellent approximations of Kolmogorov complexity.

3.2 Selection algorithms

The best selection algorithm, as reported in the study of Hemmati et al. was the *(1+1) evolutionary algorithm (EA)* [9]. Evolutionary algorithms imitate evolution as it is observed in nature, where the repeated process of recombination, mutation, and selection

leads to individuals that are increasingly adapted to their environment [7]. In evolutionary algorithms, possible solutions to the optimization task are called individuals, and a set of individuals is called a population. For example, in our context, an individual is a set of test cases. (1+1) EA is a simple variant of evolutionary algorithms that restricts the population size to just one individual. Thus, the evolution starts from a single individual, generally chosen at random, which evolves through generations. At each generation (iteration) a single offspring is generated by mutating the parent, but the offspring will not replace their parents if they have a worse fitness value. In practice this means that each test case is mutated with a probability of $1/n$, where n is the number of test cases in the individual. A mutated test case is replaced by a randomly chosen test case among the ones that are not already included. After each mutation process, the fitness value of the new population is evaluated. The fitness value is the sum of all pairwise similarities between test cases, given by the similarity function. The new generation only replace the current population if the fitness value indicates a higher degree of diversity. The evolution stops when a stop criterion is met, expressed by either a time constraint or a maximum number of iterations. Throughout the paper, we will refer to (1+1) evolutionary algorithm as the *evolutionary selection algorithm*.

Since our area of application is similar to that of Hemmati et al. (i.e., test case selection), we will also rely on *(1+1) EA*. We also chose to include in our study a simpler, more commonly used algorithm, namely *greedy-based minimization*, using it as a baseline of comparison to assess whether an evolutionary algorithm is indeed needed given its additional complexity. When selecting n test cases out of a test suite of N test cases, greedy-based minimization will remove the $N - n$ test cases with the largest pairwise similarity value. It will do so at each step of the algorithm by selecting the two test cases that have the largest similarity value. If more than one pair of test cases evaluates to the same maximum similarity value, a random one is chosen. The greedy approach continuously selects the two most similar test cases, but does not necessarily maximize the combined similarity of the $N - n$ test cases removed. Throughout the paper, we will refer to greedy-based minimization as the *greedy selection algorithm*.

4 Proposed solution

In order to define the test specifications for the system being tested, we relied on classification tree models, which is a well-known black-box specification technique [13]. The Norwegian tax department chose to use CTE-XL for their classification tree modeling. CTE-XL is a commercial tool partly built on the ideas of the category-partition method by Ostrand and Balcer [17], and is used to partition a test domain and generate a test

specification (consisting of a set of abstract test cases). In CTE-XL, we model the input domain of the system being tested as a classification tree (step 1), where all relevant distinguishing properties are captured at the desired granularity level. More specifically, properties related to the input domain that may affect the behavior of SOFIE are identified, and equivalence classes are defined for each property following the usual black-box testing strategies, such as boundary value analysis for example. Next, we generate abstract test case specifications from the model (step 2), which are valid combinations of equivalence classes in the model. Once the model is established and the abstract test cases are generated, we locate the test data available (production data in our industry context) in the system for a particular test, and match this data with the abstract test cases (partitions) from the model specification (step 3).

Let us examine an illustrative example. The upper right corner of Figure 2.1 shows a classification tree model for the functional domain A. It is a simple model containing three properties, (1) property A (number of X), with two equivalence classes: 1–4 and 5–10, (2) property B (has Z), with two equivalence classes: Yes and No, and (3) property C (number of Y), with three equivalence classes: 0, 1 and > 1. The lower part of Figure 2.1 shows the abstract test cases (partitions), which are all valid combinations of the equivalence classes in the model. The model and the corresponding partitions are the outcome of Steps 1 and 2, as described in the previous section. Table 2.1 contains an example set of available test data from a particular test. When matching the available test data with the partitions in the model, test case 1–10 will be contained in partition 3, as the value of property A is within the range of 1–4, the value of property B is Yes and the value of property C is > 1. Similarly, test case 11–20 will be contained in partition 12. With the domain model, the abstract test cases, and the set of available test data at hand, the next step is to select the test cases to execute in the regression test.

4.1 Random partition-based test case selection

In our previous study [18], we applied the following simple strategy for selecting test cases for regression testing, called random partition-based test case selection: Among the partitions containing test cases, we selected a random partition, and then a random test case within the partition. Next, we again selected a random partition among the remaining ones, and again a random test case within the partition. This process continued until all the partitions were selected. The process was then started again and test cases were selected from among the ones that had not yet been selected. The selection process stopped when the desired number of test cases were selected.

Despite being more efficient than random selection, this test case selection strategy is likely to be sub-optimal, since it selects test cases at random within each partition.

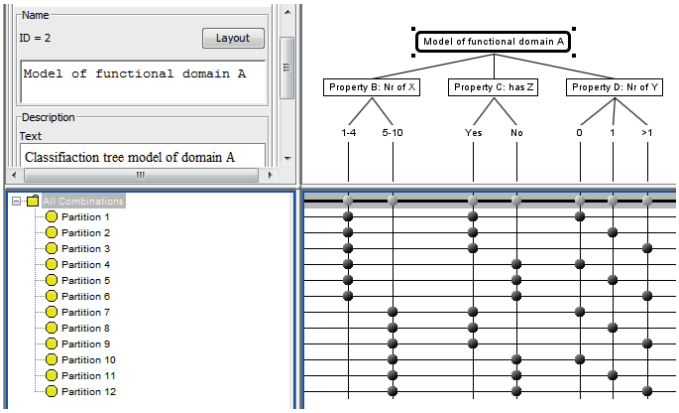


Figure 2.1: Example of a classification tree model.

Referring to the example test cases in Table 2.1, a random partition-based selection strategy would have the same likelihood of selecting test cases 1 and 2 from partition 3 as test cases 1 and 10, although intuitively it seems more likely that test case 1 and 10 will reveal different faults than test cases 1 and 2, because their property values are more diverse. Therefore, we are curious to see whether the notion of similarity measurements could improve the test case selection process, and hypothesize an improvement in the fault detection rate.

4.2 Similarity measurement for classification tree models

In order to apply similarity based test case selection, the test cases need to be encoded (1), a similarity function is needed to quantify the similarity between test cases (2), and a selection algorithm is needed to select the most diverse test cases (3).

There are two ways to encode a test case based on a classification tree model. The test case could either be encoded with its actual value for a given property, or it can be encoded as a Boolean matrix where for each equivalence class in each property, a true or false value is assigned. For example test case 1 from the example in Section 4 could either be encoded as having the values (1, Yes, 2) for property A, B, and C, respectively, or it could be encoded as (T, F, T, F, F, F, T), assigning true (T) or false (F) for all the equivalence classes in the model. The two variants of encoding are shown in Table 2.2. In order to introduce similarity-based selection within each partition, the only meaningful way to go is to use an encoding based on actual values. The Boolean encoding does not capture enough information to differentiate the various test cases within the same partition, as it is just a direct reflection of the model.

Table 2.1: Example test cases from partition 3 (1-10) and partition 12 (11-20) in Figure 2.1.

Testcase	Property B: Nr of X	Property C: Has Z	Property D: Nr of Y
Testcase 1	1	Yes	2
Testcase 2	1	Yes	2
Testcase 3	1	Yes	3
Testcase 4	2	Yes	2
Testcase 5	2	Yes	2
Testcase 6	2	Yes	4
Testcase 7	4	Yes	2
Testcase 8	4	Yes	2
Testcase 9	4	Yes	3
Testcase 10	4	Yes	4
Testcase 11	5	No	2
Testcase 12	5	No	2
Testcase 13	5	No	3
Testcase 14	7	No	2
Testcase 15	7	No	2
Testcase 16	7	No	3
Testcase 17	7	No	4
Testcase 18	8	No	2
Testcase 19	9	No	2
Testcase 20	10	No	3

Table 2.2: Test case encoding example.

(a) Example of value-based encoding of test cases.

Testcase	Property A: Nr of X	Property B: Has Z	Property C: Nr of Y
Testcase 1	1	Yes	2

(b) Example of binary encoding of test cases.

Testcase	A: 1-4	A: 5-10	B: Yes	B: No	C: 0	C: 1	C: $\gg 1$
Testcase 1	1	0	1	0	0	0	1

Given the value-based encoded test cases, a method for defining similarities between test cases is needed. Regardless of the similarity function, we have to make some choices about how to apply it for the purposes of selection. The subject model in this study does not contain any string values, but is limited to integers and boolean values. Thus, we can apply the similarity functions directly. (Further elaboration of different strategies for string matching can be found in [1].) If the property values in the model vary to a great extent, they should be normalized to prevent the small values from being obfuscated by much larger values. The output of the similarity measurement is a similarity matrix expressing the pair-wise similarity between all test cases. Having decided on how to encode the test cases and how to apply the similarity functions, we investigated two approaches of similarity-based test case selection, which will now be presented.

4.3 Pure similarity-based test case selection

One alternative is to solely rely on similarity-based test case selection strategy. By that, we mean selecting test cases from the test suite without accounting for the partitions given

from the domain model. We take the entire test suite as an input, encode the test cases, define pair-wise similarity between all the test cases in a similarity matrix, and then focus on selecting the most diverse ones. That is, given a test suite of N test cases, generate an $N \times N$ symmetric similarity matrix and select the n most diverse test cases. The set of 'most diverse' test cases will vary depending on the selection algorithm, whereas the values of the similarity matrix will vary with the similarity function. When using a pure similarity-based test case selection strategy it is not necessary to carry out steps 2 and 3 described in section 4, namely to generate the partitions and match the test data with the partitions. So rather than selecting test cases from partitions defined using human expertise, the test cases are selected purely based on similarity measures (diversity).

4.4 Similarity partition-based test case selection

Another alternative is to combine similarity measurements with partition-based selection, denoted as similarity partition-based test case selection. Rather than selecting test cases within each partition in a completely random manner, we could incorporate a similarity-based test case selection strategy within each partition. Incorporating similarity measurements within the partitions would ensure that we select as diverse test cases as possible from each partition, thus benefiting from both the model partitions and the notion of diversity within each partition. A similarity partition-based strategy generates one similarity matrix per partition containing test cases, rather than generating one for the whole test suite, and uses it to select the most diverse test cases within each partition. Applying the concept of similarity measures within each partition therefore has the advantage of keeping the similarity matrices smaller and less resource intensive.

To exemplify, when selecting four test cases from the available example test cases in Table 2.1, *random partition-based* would select two random test cases from each partition, *pure similarity-based* would select the four most diverse test cases of the entire test suite, while *similarity partition-based* would select the two most diverse test cases from each partition.

5 Experiment

This section presents an experiment aiming at investigating the use of similarity-based test case selection for the black-box regression testing of database applications. We will elaborate on the design and analysis of the experiment, report the results, and discuss the outcome.

5.1 Research questions

Our goal is to determine which similarity-based selection algorithm fares best and then assess whether and under which conditions similarity-based selection is a worthwhile and practical alternative to simpler solutions. The latter include random selection as this is by far the easiest and least expensive selection technique to apply and any alternative must be justified by significant improvements. The research questions for this study are as follows:

RQ1 For each selection algorithm, which similarity function, i.e. Euclidian, Manhattan, Mahalanobis and NCD, is best suited for defining similarity between test cases, in order to obtain the best fault detection rate and selection execution time when performing similarity-based selection of test cases generated from classification tree models?

RQ2 When used in combination with their best similarity function (RQ1), which one of the selection algorithms, i.e., evolutionary and greedy, provides better fault detection rates and selection execution times, when selecting test cases generated from classification tree models?

RQ3 Which one of the selection strategies, random partition-based, similarity partition-based and pure similarity-based (the two latter ones incorporating the best combination of similarity function —RQ1—and selection algorithm—RQ2) provides better fault detection rates and selection execution times when selecting test cases generated from classification tree models?

RQ4 Comparing the best selection approach from RQ3 and a random approach, which one provides better fault detection rates and selection execution times when selecting test cases generated from classification tree models?

5.2 Design and analysis

The subject test suite for the experiment contains 5,670 test cases, split across 130 partitions in the particular model for the domain under test. The test suite is based on actual data from the production environment in the SOFIE project, and was the only available input file for the particular regression test under consideration. For evaluation purposes we ran the regression test on the entire test suite. The regression test compares two runs of a baseline and delta version of the system under test, resulting in a set of deviations that indicate a change (regression fault or a correct change). More information on the specifics of the regression test is available in [18]. The subject test suite in this

study resulted in 522 deviations and of these, 136 deviations were faulty, capturing 15 distinct faults. All the faults are real faults (no seeded faults), emerging from a test suite in industry.

In order to determine the best similarity function and selection algorithm, we ran all combinations of similarity functions and selection algorithms for the entire test suite. We applied the similarity functions Euclidian, Manhattan, Mahalanobis and NCD to generate a similarity matrix for the entire test suite, and in turn used each of them as inputs for both the evolutionary and greedy selection algorithms. Each combination of similarity function and selection algorithm was used to select subsets of test cases, ranging from 5% to 95% of the entire test suite, with 5% intervals. For each subset we logged the exact test cases selected, and reported the number of distinct faults revealed, along with the selection execution time. We repeated the exercise 175 times in order to gain satisfactory statistical power when comparing similarity functions and selection algorithms for RQ1 and RQ2. Specifically, we scheduled the experiment to run in parallel on a cluster and the maximum number of nodes available was 175. With one parallel run on each node, this led to 175 repetitions, which was deemed sufficient to detect differences large enough to be of practical interest.

Once the outcome of RQ1 and RQ2 was concluded, the pure similarity-based approach was given directly, and constituted of the best combination of similarity function and selection algorithm. The similarity partition-based approach was composed by incorporating the best combination of similarity function and selection algorithm with the partition-based approach. Again we selected a subset of test cases ranging from 5% to 95% of the entire test suite, with 5% intervals, logged the same data points and repeated the exercise 175 times for random partition-based, similarity partition-based, pure similarity-based and random in order to address RQ3 and RQ4.

As suggested in [2] and [19], we selected a non-parametric statistical test since we could not fulfill the underlying assumption of normality and equal variance between our data samples. More specifically, we used a two-tailed Mann-Whitney U-tests (Wilcox test in R) to conduct pair-wise algorithm comparisons for all sample values, ranging from 5% to 95%. The p-values for each comparison is reported, and $\alpha = 0.05$ is used whenever referring to statistical significance. However, a Mann-Whitney U-test reports only whether there is a statistically significant difference between two algorithms, but does not clarify the magnitude of the difference. Thus, we use the $\hat{A}_{12}$ effect size measure to assess the practical significance of differences. An $\hat{A}_{12}$ effect size measurement value of 0.5 indicates that there is no difference between the two samples compared, whereas values above 0.5 indicates that sample A is superior to sample B, and opposite for values smaller than 0.5. The further away from 0.5, the larger the effect size. The value of the

effect size measurement is reported, but for increased visibility we have also categorized the effect size into Small, Medium and Large. The categories resolves to the following interpretation: Small < 0.10, $0.10 <$ Medium < 0.17 and Large > 0.17, the value being the distance from the 0.5 value [19].

5.3 Results

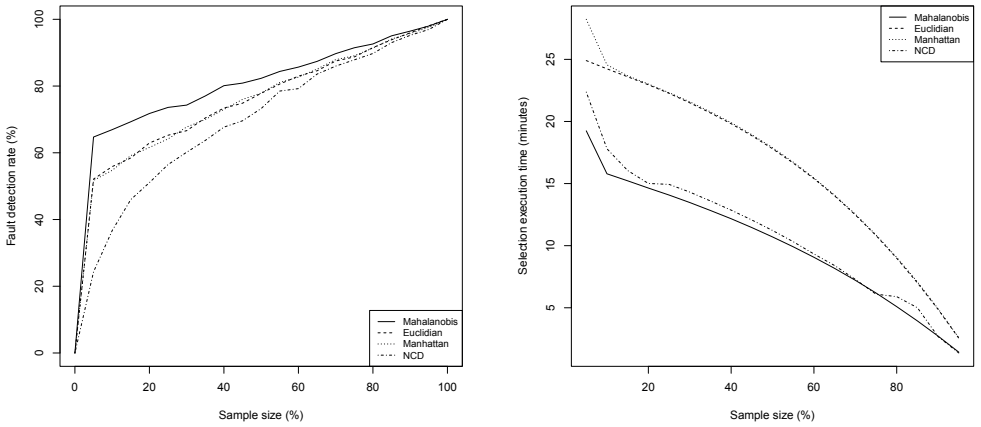
This section will report the results from testing the experimental hypotheses that can be derived from the research questions in Section 5.1.

Throughout this section the results will be reported as follows:

- A graph showing the average percentage of faults found (y-axis) per percentage of selected test cases (x-axis).
- A graph showing the average selection execution time (y-axis) in minutes per percentage of selected test cases (x-axis).
- A table showing the results of the statistical tests and effect size measurement for each algorithm comparison. The table will adhere to the following structure: Column one indicates the sample size in percentage. Then each algorithm comparison will be represented in one column, which is split into three sub-columns. The first sub-column reports the p-value from the Mann-Whitney U-test, the second sub-column reports the *superior* algorithm of the two, and the third column reports the $\hat{A}_{12}$ effect size measurement, both the actual value and on a three-point scale for helping the visualization of trends, i.e. Small, Medium or Large. The same table structure will be used to report comparisons of both fault detection rate and selection execution time.

Research question 1—Similarity functions

The results for each of the similarity functions combined with the greedy selection algorithm are depicted in Figure 2.2, Table 2.3. By looking at the graphs in Figure 2.2(a) and Figure 2.2(b), it seems evident that Mahalanobis is the best similarity function for the greedy selection algorithm, since it has the highest fault detection rate and the lowest selection execution time. This observation is confirmed by the conducted statistical tests and the measured effect size (only the results regarding fault detection rate are reported), shown in Table 2.3. In terms of fault detection rate reported in Table 2.3, Mahalanobis is significantly higher than Euclidian and Manhattan for all sample sizes up to 70% and higher, than NCD for all sample sizes. The effect size is large for smaller sample sizes, medium for medium-sized sample sizes and small for larger sample sizes, all in favor of



(a) Average number of faults found (y-axis) per number of test cases selected (x-axis) for the greedy selection algorithm.

(b) Average selection execution time (y-axis) per number of test cases selected (x-axis) for the greedy selection algorithm.

Figure 2.2: Graphs comparing different similarity functions for the greedy selection algorithm.

Mahalanobis, except for one sample value at 95%. The results also indicate that there is no significant difference in the fault detection rate between the Euclidian and Manhattan similarity functions, and there is no consistency in the effect size reported. NCD is worse than the three others for all sample sizes.

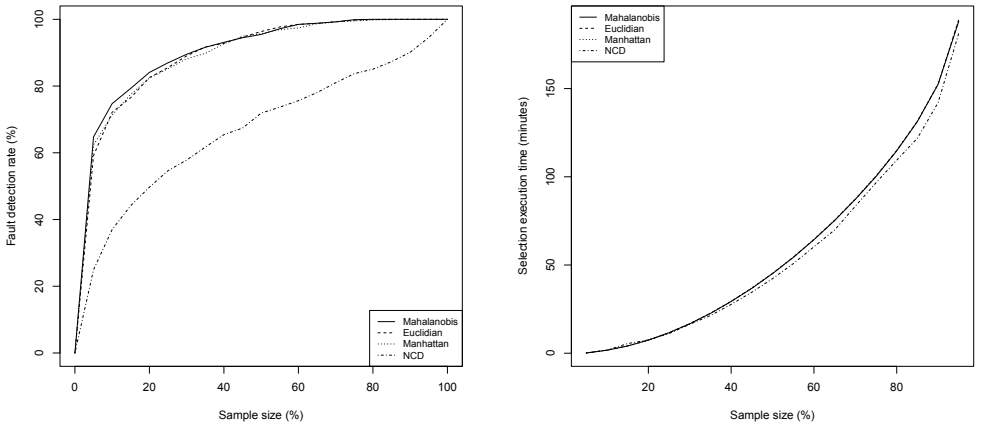
In terms of selection execution time, Mahalanobis and NCD is significantly quicker than both Euclidian and Manhattan for all sample sizes, and the effect size is large. Mahalanobis is quicker than NCD for the lower sample sizes, whereas NCD is faster for the large sample sizes. Between Euclidian and Manhattan, Euclidian shows a better selection execution time, but with varying significance and effect sizes. A general note regarding the selection execution time is that the greedy selection algorithm performs increasingly better the closer it gets to 100% of the sample size. This is because the greedy algorithm excludes the most similar test cases rather than selecting the most diverse ones, i.e. when selecting n test cases from a sample of N test cases, the greedy algorithm would exclude $N - n$ test cases rather than select n test cases.

Given the results discussed above, it is important to use the Mahalanobis similarity function in combination with the greedy selection algorithm in order to obtain the best fault detection rates and selection execution time.

Figure 2.3 and Table 2.4 depict and report the results for each of the similarity functions combined with the evolutionary selection algorithm. As the graph in Figure 2.3(b)

Table 2.3: Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing fault detection rate across the similarity functions Euclidian, Manhattan, Mahalanobis and NCD for the greedy selection algorithm.

Comparison / Sample size (%)	Mahalanobis (A), Euclidian (B)			Mahalanobis (A), Manhattan (C)			Euclidian (B), Manhattan (C)		
	p-value	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**
5	< 0.0001	A	Large (0.9207)	< 0.0001	A	Large (0.9392)	0.7158	B	Small (0.4894)
10	< 0.0001	A	Large (0.8803)	< 0.0001	A	Large (0.9037)	0.1466	B	Small (0.4572)
15	< 0.0001	A	Large (0.8528)	< 0.0001	A	Large (0.8423)	0.4149	C	Small (0.5241)
20	< 0.0001	A	Large (0.7840)	< 0.0001	A	Large (0.8284)	0.1388	B	Small (0.4561)
25	< 0.0001	A	Large (0.7645)	< 0.0001	A	Large (0.7931)	0.1459	B	Small (0.4568)
30	< 0.0001	A	Large (0.7557)	< 0.0001	A	Large (0.7117)	0.1953	C	Small (0.5389)
35	< 0.0001	A	Large (0.6969)	< 0.0001	A	Large (0.7109)	0.7047	B	Small (0.4886)
40	< 0.0001	A	Large (0.7142)	< 0.0001	A	Large (0.7271)	0.4939	B	Small (0.4795)
45	< 0.0001	A	Large (0.6871)	< 0.0001	A	Medium (0.6551)	0.2968	C	Small (0.5315)
50	< 0.0001	A	Medium (0.6444)	< 0.0001	A	Medium (0.6343)	0.8441	C	Small (0.5059)
55	< 0.0001	A	Medium (0.6243)	0.0003	A	Medium (0.6088)	0.5831	C	Small (0.5165)
60	0.0033	A	Small (0.5879)	0.0001	A	Medium (0.6137)	0.6210	B	Small (0.4852)
65	0.0011	A	Small (0.5974)	0.0147	A	Small (0.5732)	0.4950	C	Small (0.5205)
70	0.0053	A	Small (0.5829)	0.0459	A	Small (0.5593)	0.5272	C	Small (0.5190)
75	0.0020	A	Small (0.5920)	0.0030	A	Small (0.5878)	0.7378	C	Small (0.5099)
80	0.1215	A	Small (0.5457)	0.1228	A	Small (0.5456)	0.9727	B	Small (0.4990)
85	0.0356	A	Small (0.5602)	0.1770	A	Small (0.5385)	0.4723	C	Small (0.5207)
90	0.3730	A	Small (0.5244)	0.0550	A	Small (0.5532)	0.3218	B	Small (0.4723)
95	0.5088	B	Small (0.4845)	0.3755	A	Small (0.5218)	0.1308	B	Small (0.4636)
	Mahalanobis (A), NCD (D)			**Euclidian (B), NCD (D)**			**Manhattan (C), NCD (D)**		
	p-value	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**
5	< 0.0001	A	Large (1)	< 0.0001	B	Large (0.9970)	< 0.0001	C	Large (0.9967)
10	< 0.0001	A	Large (0.9991)	< 0.0001	B	Large (0.9515)	< 0.0001	C	Large (0.9440)
15	< 0.0001	A	Large (0.9879)	< 0.0001	B	Large (0.8509)	< 0.0001	C	Large (0.8655)
20	< 0.0001	A	Large (0.9509)	< 0.0001	B	Large (0.8227)	< 0.0001	C	Large (0.7952)
25	< 0.0001	A	Large (0.9038)	< 0.0001	B	Large (0.7647)	< 0.0001	C	Large (0.7282)
30	< 0.0001	A	Large (0.8831)	< 0.0001	B	Large (0.7001)	< 0.0001	C	Large (0.7297)
35	< 0.0001	A	Large (0.8564)	< 0.0001	B	Large (0.6922)	< 0.0001	C	Large (0.6967)
40	< 0.0001	A	Large (0.8318)	< 0.0001	B	Medium (0.6651)	< 0.0001	C	Medium (0.6402)
45	< 0.0001	A	Large (0.8114)	< 0.0001	B	Medium (0.6543)	< 0.0001	C	Large (0.6821)
50	< 0.0001	A	Large (0.7460)	< 0.0001	B	Medium (0.6342)	< 0.0001	C	Medium (0.6379)
55	< 0.0001	A	Large (0.6879)	0.0366	B	Small (0.5650)	0.0145	C	Small (0.5760)
60	< 0.0001	A	Large (0.7245)	0.0003	B	Medium (0.6116)	0.0002	C	Medium (0.6125)
65	< 0.0001	A	Medium (0.6306)	0.2085	B	Small (0.5390)	0.0912	C	Small (0.5524)
70	< 0.0001	A	Medium (0.6389)	0.1165	B	Small (0.5483)	0.0275	C	Small (0.5682)
75	< 0.0001	A	Medium (0.6355)	0.2853	B	Small (0.5330)	0.1071	C	Small (0.5494)
80	0.0014	A	Small (0.5978)	0.0357	B	Small (0.5644)	0.0621	C	Small (0.5573)
85	0.0058	A	Small (0.5823)	0.4329	B	Small (0.5235)	0.0813	C	Small (0.5521)
90	0.0101	A	Small (0.5740)	0.1434	B	Small (0.5425)	0.5825	C	Small (0.5160)
95	0.0357	A	Small (0.5547)	0.0040	B	Small (0.5734)	0.2820	C	Small (0.5286)



(a) Average number of faults found (y-axis) per number of test cases selected (x-axis) for the evolutionary selection algorithm

(b) Average selection execution time (y-axis) per number of test cases selected (x-axis) for the evolutionary selection algorithm

Figure 2.3: Graphs comparing different similarity functions for the evolutionary selection algorithm.

show, there is no practical difference between the similarity functions regarding selection execution time (the lines are virtually on top of each other). The same accounts for the fault detection rate between Euclidian, Manhattan and Mahalanobis, while NCD falls short of the others, as shown in Figure 2.3(a). The statistical tests (only the results regarding fault detection rate are reported) confirm that Euclidian, Manhattan and Mahalanobis are better than NCD for all sample sizes and the effect size is large. Between the former three, the null-hypothesis stating that there is no difference in fault detection rates between the algorithms cannot be rejected other than for the sample sizes 5% to 15% in the Mahalanobis vs. Euclidian comparison and for the sample sizes 5%, 10%, 25%, 35%, 60%, 75% and 80% in the Mahalanobis vs. Manhattan comparison, all in favor of Mahalanobis. Regarding selection execution time, there is no notable difference among algorithms. The results for the evolutionary selection algorithm do not show clear differences as for the greedy approach, except for the poor results of NCD, but if we have to choose one, Mahalanobis is again the preferred similarity function. Despite not being superior for all sample sizes, Mahalanobis is significantly better for smaller sample sizes, which are the most important ones, and it is not inferior to the others for any other sample sizes.

In summary, we can address RQ1 by stating that Mahalanobis should be the preferred similarity function for both the greedy and evolutionary selection algorithms. This

Table 2.4: Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing fault detection rates across the similarity functions Euclidian, Manhattan, Mahalanobis and NCD for the evolutionary selection algorithm.

Comparison / Sample size (%)	Mahalanobis (A), Euclidian (B)			Mahalanobis (A), Manhattan (C)			Euclidian (B), Manhattan (C)		
	p-value	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**
5	< 0.0001	A	Large (0.6797)	0.0134	A	Small (0.5740)	0.0001	C	Medium (0.6158)
10	0.0033	A	Small (0.5885)	0.0003	A	Medium (0.6103)	0.3552	B	Small (0.4721)
15	0.0042	A	Small (0.5863)	0.0515	A	Small (0.5586)	0.3309	C	Small (0.5293)
20	0.2229	A	(Small (0.5366)	0.0526	A	Small (0.5579)	0.5269	B	Small (0.4810)
25	0.1251	A	Small (0.5459)	0.0253	A	Small (0.5668)	0.5279	B	Small (0.4811)
30	0.5644	A	Small (0.5171)	0.1176	A	Small (0.5467)	0.2847	B	Small (0.4682)
35	0.9075	B	Small (0.4966)	0.0349	A	Small (0.5626)	0.0184	B	Small (0.4307)
40	0.6634	A	Small (0.5126)	0.4593	A	Small (0.5215)	0.7691	B	Small (0.4914)
45	0.7801	B	Small (0.4920)	0.4388	C	Small (0.4778)	0.6201	C	Small (0.5142)
50	0.1646	B	Small (0.4612)	0.6034	C	Small (0.4854)	0.3979	B	Small (0.4766)
55	0.2499	B	Small (0.4705)	0.8279	A	Small (0.5057)	0.1829	B	Small (0.4658)
60	1	None	NO effect (0.5)	0.0039	A	Small (0.5689)	0.0039	B	Small (0.4311)
65	0.7087	A	Small (0.5075)	0.5046	A	Small (0.5136)	0.7657	B	Small (0.4938)
70	0.8927	A	Small (0.5022)	0.7486	A	Small (0.5054)	0.8516	B	Small (0.4968)
75	0.2006	A	Small (0.5114)	0.0106	A	Small (0.5286)	0.1681	B	Small (0.4829)
80	0.1579	A	Small (0.5057)	0.0247	A	Small (0.5143)	0.2536	B	Small (0.4914)
85	NaN	None	NO effect (0.5)	0.0830	A	Small (0.5086)	0.0830	B	Small (0.4914)
90	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)
95	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)
	Mahalanobis (A), NCD (D)			**Euclidian (B), NCD (D)**			**Manhattan (C), NCD (D)**		
	p-value	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**	**p-value**	**Superior**	**Effect size**
5	< 0.0001	A	Large (0.9997)	< 0.0001	B	Large (0.9994)	< 0.0001	C	Large (0.9998)
10	< 0.0001	A	Large (0.9994)	< 0.0001	B	Large (0.9995)	< 0.0001	C	Large (0.9991)
15	< 0.0001	A	Large (0.9997)	< 0.0001	B	Large (0.9979)	< 0.0001	C	Large (0.9990)
20	< 0.0001	A	Large (0.9990)	< 0.0001	B	Large (0.9965)	< 0.0001	C	Large (0.9995)
25	< 0.0001	A	Large (0.9983)	< 0.0001	B	Large (0.9956)	< 0.0001	C	Large (0.9956)
30	< 0.0001	A	Large (0.9980)	< 0.0001	B	Large (0.9964)	< 0.0001	C	Large (0.9961)
35	< 0.0001	A	Large (0.9957)	< 0.0001	B	Large (0.9964)	< 0.0001	C	Large (0.9899)
40	< 0.0001	A	Large (0.9986)	< 0.0001	B	Large (0.9991)	< 0.0001	C	Large (0.9991)
45	< 0.0001	A	Large (0.9967)	< 0.0001	B	Large (0.9978)	< 0.0001	C	Large (0.9968)
50	< 0.0001	A	Large (0.9890)	< 0.0001	B	Large (0.9937)	< 0.0001	C	Large (0.9887)
55	< 0.0001	A	Large (0.9977)	< 0.0001	B	Large (0.9988)	< 0.0001	C	Large (0.9937)
60	< 0.0001	A	Large (0.9964)	< 0.0001	B	Large (0.9964)	< 0.0001	C	Large (0.9937)
65	< 0.0001	A	Large (0.9909)	< 0.0001	B	Large (0.9916)	< 0.0001	C	Large (0.9906)
70	< 0.0001	A	Large (0.9898)	< 0.0001	B	Large (0.9918)	< 0.0001	C	Large (0.9902)
75	< 0.0001	A	Large (0.9985)	< 0.0001	B	Large (0.9965)	< 0.0001	C	Large (0.9936)
80	< 0.0001	A	Large (0.9885)	< 0.0001	B	Large (0.9870)	< 0.0001	C	Large (0.9846)
85	< 0.0001	A	Large (0.9454)	< 0.0001	B	Large (0.9454)	< 0.0001	C	Large (0.9425)
90	< 0.0001	A	Large (0.9310)	< 0.0001	B	Large (0.9310)	< 0.0001	C	Large (0.9310)
95	< 0.0001	A	Large (0.8218)	< 0.0001	B	Large (0.8218)	< 0.0001	C	Large (0.8218)

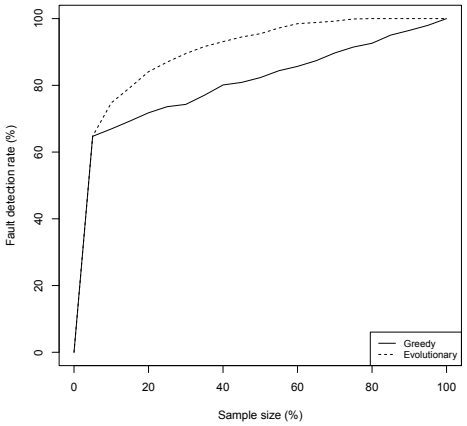
could indicate that it is important to consider correlation among model properties, since incorporation of correlation is the main differentiating factor between the Mahalanobis similarity function and the two simpler similarity functions Euclidian and Manhattan. NCD falls short of the others in terms of the fault detection rate. The most plausible reason is that the test cases are represented by a fairly simple structure that accounts for very little information distance between their compressed versions. Thus, NCD is not able to pick up the minor differences as well as simpler geometrical functions.

Research question 2—Selection algorithms

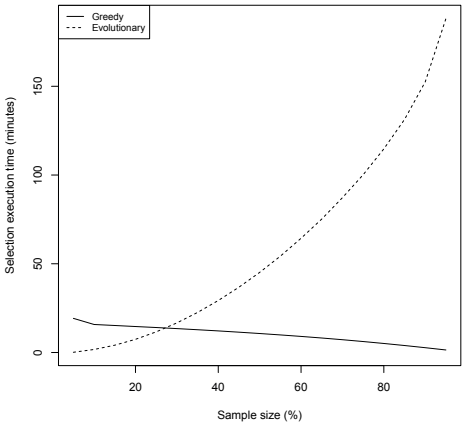
Figure 2.4 and Table 2.5 depict and report the results for the greedy and evolutionary selection algorithms, each combined with their best similarity function (Mahalanobis). By looking at the graphs, it seems obvious that evolutionary is better than greedy in terms of fault detection rate, whereas their selection execution time follow different patterns. The selection execution time decreases for the greedy approach as the sample size increases (for the reasons mentioned in Section 5.3), whereas it increases for the evolutionary approach. The cross-over point is between sample size 25% and 30%. Regarding fault detection rates, the results reported from the statistical tests and effect size measures show that evolutionary is better than greedy for all sample sizes except 5%, where there is no statistical significance. The effect size is large for all sample sizes except 5% (small) and 95% (medium), still in favor of the evolutionary approach. Also important is the result that evolutionary converges to 100% fault detection much faster than the greedy approach, i.e. it reaches 98.5% and 100% detection rate at 60% and 80% sample sizes, respectively. In terms of selection execution time there is a significant difference for all sample values, in favor of the evolutionary approach for 5% to 25% sample sizes, and in favor of the greedy approach for 30% to 95% sample sizes. Given the superiority of the evolutionary approach in terms of fault detection rate, along with the fact that it performs better for smaller sample sizes, the outcome of RQ2 is that the evolutionary selection algorithm in combination with the Mahalanobis similarity function is the best similarity-based combination for selecting test cases generated from classification tree models in our experiment.

Research question 3—Selection strategies

For RQ3, we are interested in comparing random partition-based with similarity partition-based and pure similarity-based, the latter two incorporating the best combination of similarity function and selection algorithms from RQ2, namely the evolutionary selection algorithm and the Mahalanobis similarity function. Figures 2.5(a) and 2.5(b), and Tables 2.6 and 2.7 depict and report the results for the similarity partition-based, random



(a) Average number of faults found (y-axis) per number of test cases selected (x-axis) for the best greedy and best evolutionary selection algorithm.



(b) Average selection execution time (y-axis) per number of test cases selected (x-axis) for the best greedy and best evolutionary selection algorithm.

Figure 2.4: Graphs comparing the best greedy and evolutionary selection algorithm.

Table 2.5: Data reported from Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing the greedy and evolutionary selection algorithms.

(a) Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing fault detection rate across the selection algorithms greedy and evolutionary, each combined with their best similarity function.

Comparison / Sample size (%)	Evolutionary (A), Greedy (B)		
	p-value	Superior	Effect size
5	0.9308	A	Small (0.5026)
10	< 0.0001	A	Large (0.7650)
15	< 0.0001	A	Large (0.8191)
20	< 0.0001	A	Large (0.8561)
25	< 0.0001	A	Large (0.8677)
30	< 0.0001	A	Large (0.9240)
35	< 0.0001	A	Large (0.9049)
40	< 0.0001	A	Large (0.8962)
45	< 0.0001	A	Large (0.9085)
50	< 0.0001	A	Large (0.9027)
55	< 0.0001	A	Large (0.9180)
60	< 0.0001	A	Large (0.9431)
65	< 0.0001	A	Large (0.9164)
70	< 0.0001	A	Large (0.9007)
75	< 0.0001	A	Large (0.8720)
80	< 0.0001	A	Large (0.8457)
85	< 0.0001	A	Large (0.7886)
90	< 0.0001	A	Large (0.7057)
95	< 0.0001	A	Medium (0.6371)

(b) Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing selection execution times across the selection algorithms greedy and evolutionary, each combined with their best similarity function.

Comparison / Sample size (%)	Evolutionary (A), Greedy (B)		
	p-value	Superior	Effect size
5	< 0.0001	A	Large (0)
10	< 0.0001	A	Large (0)
15	< 0.0001	A	Large (0)
20	< 0.0001	A	Large (0)
25	< 0.0001	A	Large (0.0020)
30	< 0.0001	B	Large (0.9974)
35	< 0.0001	B	Large (1)
40	< 0.0001	B	Large (1)
45	< 0.0001	B	Large (1)
50	< 0.0001	B	Large (1)
55	< 0.0001	B	Large (1)
60	< 0.0001	B	Large (1)
65	< 0.0001	B	Large (1)
70	< 0.0001	B	Large (1)
75	< 0.0001	B	Large (1)
80	< 0.0001	B	Large (1)
85	< 0.0001	B	Large (1)
90	< 0.0001	B	Large (1)
95	< 0.0001	B	Large (1)

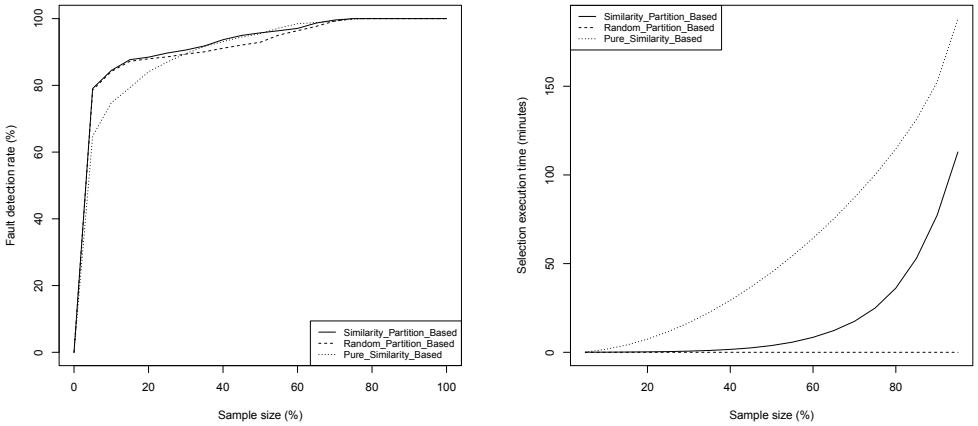
partition-based and pure similarity-based approaches. It is clear from the graph that combining similarity measurements with a partition-based approach offers improved selection execution times when compared with a pure similarity-based approach. By using a partition-based strategy, the selection problem is divided into smaller sub-problems. Consequently, the similarity matrices are smaller and easier to work with, e.g. for the 30% sample size, the similarity partition-based approach uses less than one minute, whereas pure similarity-based uses 22 minutes, which is a substantial improvement in practice. In terms of the fault detection rate, the similarity partition-based curve is steeper than pure similarity-based at the beginning, thus reaching a higher fault detection rate at an earlier stage (for smaller sample sizes). Additionally, it converges even faster to 100% fault detection (at a 75% sample size) and is more reliable since it shows lower variance around the mean (not reported explicitly in the graph). The fault detection rate of the similarity partition-based approach is significantly better than the pure similarity-based approach for smaller sample sizes up to 25%, with a large effect size for 5% to 20%. However, the pure similarity-based approach is significantly better than the similarity partition-based approach for a 60% sample size with a medium effect size. The selection execution time is significantly different for all sample sizes, with a large effect size in favor of the similarity partition-based approach.

The difference between the similarity partition-based and random partition-based is more marginal. The selection execution time is in favor of random partition-based, with significant differences for all sample sizes. However, for sample values up to 50%, execution time ranges from near 0 to up to 4 minutes for similarity partition-based, as opposed to less than a second for random partition-based, a difference which is not of great practical significance. In terms of the fault detection rate, similarity partition-based is significantly better than random partition-based for sample sizes of 15%, between 25% and 55% and for 65%, and is superior for all other sample sizes, though with modest improvements.

To address RQ3, when selecting test cases generated from classification tree models, similarity partition-based provides overall the best fault detection rates, whereas random partition-based shows the lowest selection execution time. It is, however, worth pointing out that the practical significance, which in the end is what decides the usefulness of the approach, is clearly in favor of the prior candidate solution since the selection execution time only comes into play with extremely large test suites.

Research question 4—Compared with a random approach

Figure 2.6 and Table 2.8 depict and report the results we obtained when comparing similarity partition-based and random test case selection approaches. In terms of fault detection, the improvements when adopting a more sophisticated test case selection strat-



(a) Average number of faults found (y-axis) per number of test cases selected (x-axis) for the three best approaches.

(b) Average selection execution time (y-axis) per number of test cases selected (x-axis) for the three best approaches.

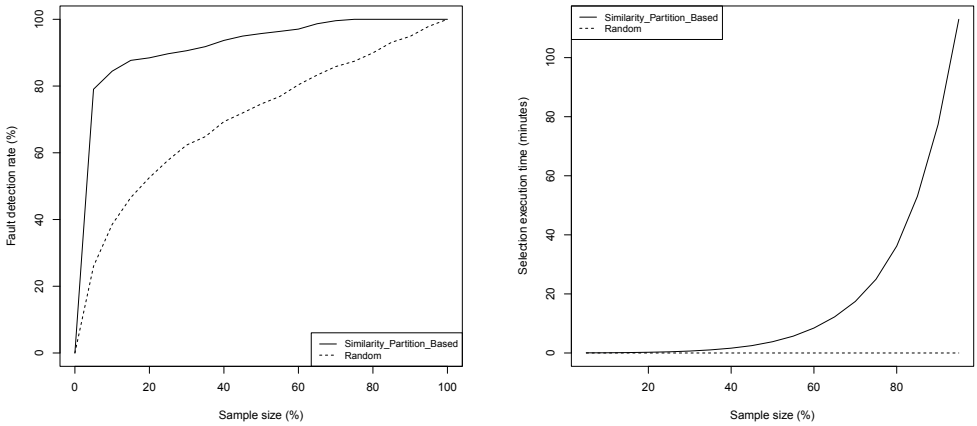
Figure 2.5: Graphs comparing the three best approaches.

Table 2.6: Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing fault detection rates across the similarity partition-based, random partition-based and pure similarity-based test case selection strategies.

Comparison / Sample size (%)	Similarity Partition-Based (A), Random Partition-Based (B)			Similarity Partition-Based (A), Pure Similarity-Based (C)			Random Partition-Based (B), Pure Similarity-Based (C)		
	p-value	Superior	Effect size	p-value	Superior	Effect size	p-value	Superior	Effect size
5	0.4677	A	Small (0.5199)	< 0.0001	A	Large (0.9169)	< 0.0001	B	Large (0.9159)
10	0.4594	A	Small (0.5199)	< 0.0001	A	Large (0.8310)	< 0.0001	B	Large (0.8225)
15	0.0428	A	Small (0.5345)	< 0.0001	A	Large (0.8110)	< 0.0001	B	Large (0.7992)
20	0.0984	A	Small (0.5367)	< 0.0001	A	Large (0.6716)	< 0.0001	B	Medium (0.6537)
25	0.0023	A	Small (0.5776)	0.0019	A	Small (0.5895)	0.1624	B	Small (0.5395)
30	0.0017	A	Small (0.5852)	0.3066	A	Small (0.5296)	0.3330	C	Small (0.4722)
35	< 0.0001	A	Medium (0.6141)	0.7949	C	Small (0.4925)	0.0019	C	Small (0.4104)
40	< 0.0001	A	Medium (0.6624)	0.6126	A	Small (0.5138)	< 0.0001	C	Medium (0.3827)
45	< 0.0001	A	Large (0.6715)	0.9072	C	Small (0.4968)	< 0.0001	C	Medium (0.3641)
50	< 0.0001	A	Medium (0.6600)	0.5342	C	Small (0.4829)	< 0.0001	C	Medium (0.3539)
55	0.0109	A	Small (0.5697)	0.0067	C	Small (0.4269)	< 0.0001	C	Medium (0.3700)
60	0.2574	A	Small (0.5306)	< 0.0001	C	Medium (0.3863)	< 0.0001	C	Medium (0.3650)
65	0.0058	A	Small (0.5651)	0.3748	C	Small (0.4817)	0.0004	C	Small (0.4181)
70	0.2287	A	Small (0.5175)	0.1681	A	Small (0.5204)	0.8613	B	Small (0.5028)
75	NaN	None	NO effect (0.5)	0.0830	A	Small (0.5086)	0.0830	B	Small (0.5085)
80	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)
85	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)
90	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)
95	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)	NaN	None	NO effect (0.5)

Table 2.7: Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing selection execution time across the similarity partition-based, random partition-based and pure similarity-based test case selection strategies.

Comparison / Sample size (%)	Similarity Partition-Based (A), Random Partition-Based (B)			Similarity Partition-Based (A), Pure Similarity-Based (C)			Random Partition-Based (B), Pure Similarity-Based (C)		
	p-value	Superior	Effect size	p-value	Superior	Effect size	p-value	Superior	Effect size
5-95	< 0.0001	B	Large (1)	< 0.0001	A	Large (0)	< 0.0001	B	Large (0)



(a) Average number of faults found (y-axis) per number of test cases selected (x-axis) for the similarity partition-based test case selection approach and a random selection.

(b) Average selection execution time (y-axis) per number of test cases selected (x-axis) for the similarity partition-based test case selection approach and a random selection.

Figure 2.6: Graphs comparing similarity partition-based test case selection and random selection.

egy such as similarity partition-based are highly significant compared to using a random test selection strategy. As the graph shows, at a sample size of 5%, random selection would on average identify 25% of the faults, whereas similarity partition-based selections would reveal nearly 80% of the faults. This is a considerable efficiency gain in terms of early fault detection. The results show that similarity partition-based has a significantly better fault detection rate than the random approach for all sample sizes, with a large effect size, whereas the opposite is true for selection execution time. However, as long as the selection execution time is within a satisfactory range, the fault detection rate is obviously the most important criterion of the two. We can reasonably consider the selection execution time for similarity partition-based to be acceptable, particularly since we target smaller sample sizes of selection, in which the execution takes less than four minutes for a quite large original test suite. So, in order to address RQ4, similarity partition-based offers very strong advantages over the random approach when selecting test cases generated from classification tree models.

5.4 Discussion and Further Analysis

Although the idea of incorporating similarity-based test case selections within each partition seems intuitive and promising, especially given some of the recent results of its

Table 2.8: Data reported from Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing similarity partition-based test case selection and random selection.]

(a) Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing fault detection rate across the similarity partition-based test case selection approach and a random selection.

Comparison / Sample size (%)	Evolutionary (A), Random (B)		
	P-value	**Superior**	**Effect size**
5% - 80%	< 0.0001	A	Large (> 0.9200)
85% - 90%	< 0.0001	A	Large (> 0.7800)
95%	< 0.0001	A	Medium (0.6371)

(b) Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing selection execution time across the similarity partition-based test case selection approach and a random selection.

Comparison / Sample size (%)	Evolutionary (A), Random (B)		
	p-value	**Superior**	**Effect size**
5% - 95%	< 0.0001	B	Large (1)
85% - 90%	< 0.0001	B	Large (1)
95%	< 0.0001	B	Large (1)

application in other contexts [9], the overall results we obtained are only marginally better when compared to a random selection within each partition. We analyzed the data more carefully to find plausible explanations for this unexpected result. It turns out that many of the faults are located in partitions containing relatively few test cases. Hence, they would surface quickly with a partition-based approach, regardless of whether the selection was random or similarity-based within each partition. Such situations are expected to occur when testing is driven by an operational or usage profile, and faults are uncovered when executing unusual scenarios for which relatively few test cases are defined [16]. Though we have not used an operational profile, our original test suite is based on real taxpayer data, and consequently we have more test cases representing common scenarios that are well understood and less faulty, whereas we have fewer test cases accounting for more unusual situations where faults are more likely to hide. In situations when faults are only contained in small partitions, the impact of similarity-based selection within partitions becomes negligible and should not be applied, especially on very large test suites where selection execution time matters.

In our context, some of the faults were located in partitions containing many test cases. To be specific, 73% of the faults were located in small partitions only containing up to three test cases, whereas 27% of the faults were located in larger partitions of up to 340 test cases. As expected, the fault detection rate for the faults from the small partitions do not differ between similarity partition-based and random partition-based. So the main source of differences between random partition-based and similarity partition-based lies in the detection rate of the faults from the larger partitions. A detailed analysis shows that for all the larger partitions containing faults, the fault detection rate was significantly higher for similarity partition-based than random partition-based. And the larger the partition, the larger the impact of similarity measurement. As an example, fault X is contained in a partition with five test cases. In this case, similarity partition-based selection detected the fault in 112 out of 175 runs for the 5% selection sample, while the results for random partition-based was 111/175. For all other sample sizes, both selection strategies showed a

Selection strategy \ Sample size (%)	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75-100
Similarity Partition-Based	2%	5%	8%	13%	23%	29%	39%	53%	62%	68%	73%	78%	90%	97%	100%
Random Partition-Based	1%	3%	4%	10%	14%	17%	25%	33%	41%	47%	63%	73%	83%	95%	100%

Table 2.9: The average detection rate for Fault Z and W that are contained in a large partition. Each of the faults are present in one single test case among 340 test cases in the partition.

100% detection for this fault. This is a marginal improvement, which can be explained by the fact that the partition still contains few test cases. Fault Y is contained in a partition with 18 test cases. The partition is a bit larger than the previous one, and so is the fault detection improvement of similarity partition-based over random partition-based. For the 5% and 10% selection samples, the detection rate of fault Y is 31/175 vs. 23/175 and 99/175 vs. 98/175, both in favor of similarity partition-based, respectively. It is 100% for all other sample sizes. For the 5% sample, the difference is of practical significance. Faults Z and W are contained in the same partition constituting a total of 340 test cases. The fault detection rate for these faults combined are shown in Table 2.9. As the table shows, the similarity partition-based selection yields a better detection rate for all sample sizes, and the difference is practically significant for most sample sizes. By practically significant we mean that the relative increase in fault detection rate is considerable, i.e. 30% to 100% improvement up to 50% sample size. Table 2.10 also shows that the results are statistically significant (p-values are below our significance threshold) for most sample sizes with the effect size varying from small to large. The above results suggests that similarity partition-based should be preferred when faults are located in partitions containing a large number of test cases. This is often the case, for example, in safety critical systems where the most critical or complex scenarios and components tend to be more exercised by testing.

Another important point to highlight regarding the overall marginal improvement of similarity partition-based over random partition-based test case selection in this study, is the lack of variability in the subject model. To benefit significantly from similarity measurements when selecting test cases within partitions, the equivalence classes must be defined in such a way that they contain significant variation. If all model properties are defined as either having constant values (i.e. green, red or blue) or boolean values, there is little room left for test case diversity within each partition. In such cases all test cases within a partition would have the same encoding, and subsequently be equal, unless the distance function is expanded to define the distance between constants as well, e.g. blue is closer to green than red. Thus, an important prerequisite for using a similarity partition-based test case selection for classification tree models is to have at least one equivalence class for a property defined as a numerical range (i.e. 1–100 and $>$ 100) or a string.

Table 2.10: Mann-Whitney U-tests and $\hat{A}_{12}$ effect size measurements when comparing detection rate for fault Z and W across the similarity partition-based and random partition-based test case selection strategies.

Comparison / Sample size (%)	Evolutionary (A), Greedy (B)		
	p-value	**Superior**	**Effect size**
5	0.5586	A	Small (0.5057)
10	0.1757	A	Small (0.52)
15	0.0428	A	Small (0.5345)
20	0.0984	A	Small (0.5367)
25	0.0023	A	Small (0.5776)
30	< 0.0001	A	Medium (0.6059)
35	< 0.0001	A	Medium (0.6141)
40	< 0.0001	A	Medium (0.6624)
45	< 0.0001	A	Large (0.6715)
50	< 0.0001	A	Medium (0.66)
55	0.0068	A	Small (0.5736)
60	0.2573	A	Small (0.5306)
65	0.0058	A	Small (0.5651)
70	0.2287	A	Small (0.5175)
75	NaN	None	NO effect (0.5)
80	NaN	None	NO effect (0.5)
85	NaN	None	NO effect (0.5)
90	NaN	None	NO effect (0.5)
95	NaN	None	NO effect (0.5)

The more diversity within partitions, the more likely our approach is to benefit from similarity partition-based test case selection. The subject model in our study contained 11 properties at the bottom level, split up into 26 equivalence classes, 21 of them being constants, whereas 5 were defined as integer ranges. The magnitude of the ranges in equivalence classes were limited, i.e. 2–3, 2–5 and 2–9. Despite the limited variation enabled within the partitions, the results show that the fault detection rate improved (though to a limited extent) for all sample sizes when using similarity partition-based test case selections. The results were not significant for all sample sizes, nor was the effect size constantly large, but still the average fault detection rate was better for all sample values until a 100% detection rate was reached. Though more studies are required, our results suggest that a classification tree model accounting for even more variability than in our experiment, would be likely to yield further benefits from a similarity partition-based test case selection.

To summarize the discussion, the level of variability and number of test cases within partitions are two important factors affecting the extent of the benefit that can be expected from similarity partition-based test case selection. In general, when modeling large complex systems in an industrial setting, it is likely that the equivalence classes defined in the model capture a variety of possible system values, as the model is a high level representation of the system. Consequently, many equivalence classes in the model will

represent a range of possible system values and induce a great deal of variability within each partition. When executing test cases based on live system data, the distribution of test cases across partitions will vary, but many partitions are likely to contain large numbers of test cases. We recommend similarity partition-based test case selection under these conditions, while simpler solutions, such as random partition-based, are sufficient otherwise. One could also combine the two by defining a threshold value on the number of test cases per partition, and only use similarity based selection above this threshold.

5.5 Threats to validity

The fact that the study only includes one test suite derived from one classification tree model is a threat to the external validity of the study. Ideally, we should have included one or more additional test suites in the experiment, while varying model complexity and variability. But running and evaluating large test suites in real industry projects is a comprehensive and costly operation and the one test suite used in our study is large and based on operational data. It also triggers failures based on real faults detected in an operational system.

Throughout the experiment we compared several algorithms, by conducting multiple pairwise comparisons. This inflates the probability of a Type I error (reject the null hypothesis when it is true), which is a threat to conclusion validity. This could be adjusted by using, for example, *Bonferroni adjustment*. However, as reported by Arcuri and Briand (2011) [2], the Bonferroni adjustment has been repeatedly criticized in the literature. For example a serious problem associated with the standard Bonferroni procedure is a substantial reduction in the statistical power of rejecting an incorrect null hypothesis. So rather than performing adjustments, we have instead chosen to report all p-values from all the statistical tests. The results are thus transparent and the readers have all the data at hand to form an opinion for themselves and to verify the conclusions we drew from our tests.

6 Conclusion and future work

Within the context of regression testing, we proposed a new strategy for selecting test cases generated from classification tree models, a well-known test generation strategy for black-box testing. Such an approach is a particularly useful alternative in an environment where source code analysis is either not convenient, scalable, or even possible. In short, our selection strategy selects, in a balanced way, test cases from all input partitions defined by the classification tree, while attempting to select the most diverse test cases from each partition. We conducted an experiment in an industrial setting–a large and critical

database application developed by the Norwegian tax department–to determine which similarity-based selection algorithm, i.e., similarity function and selection algorithm, fared best for selecting test cases in a large regression test suite. We also compared our approach (coined similarity partition-based selection) against pure random selection and random selection within partitions. We compared both fault detection rate and selection execution time. In general random selection is superior to similarity-based selection in terms of selection execution time. However, the difference for smaller sample sizes in the range of interest is less than a few minutes (i.e., 39 seconds when selecting 30% of the test suite when comparing similarity partition-based with random selection). Given the limited practical consequences of the difference in selection execution time, we have used fault detection rate as the main criterion for comparison.

Among all alternatives, combining the Mahalanobis similarity function and the 1+1 EA algorithm proved to be the most efficient with regards to fault detection rate. The experiment also assessed whether similarity-based selection is a worthwhile and practical alternative to simpler solutions. Similarity partition-based test case selection offers far better fault detection rates compared to a random selection of test cases. For example, by selecting 5% of the test cases in a test suite, the fault detection rate of similarity partition-based is nearly 80%, as opposed to 25% for random selection. Despite a dramatical improvement over random, applying similarity-based selection within each partition only marginally improves, on average, the fault detection rate over random selection within each partition. Though similarity partition-based selection has better average fault detection rates for all sample sizes compared to random partition-based selection, the results are overall not statistically significant for all sample size values, and the measured effect size is in general low (i.e., low practical significance). The two most plausible explanations for these results are that (1) many of the faults in our study are located in partitions containing few test cases, along with the fact that (2) the subject classification tree model is such that diversity is often low within partitions, thus limiting the potential benefit of similarity-based test case selection. The results thus clearly suggest conditions under which similarity-based test selection is worth combining with a partition-based strategy. Consistent with such explanations, a more detailed analysis clearly shows that, for faults within large partitions, fault detection rates are significantly higher when using similarity over random within partition-based selection. This suggests that it would be beneficial to use similarity-based selection within partitions when these conditions are present, even when the increase in fault detection rate is obtained at the expense of higher execution time.

A possible future addition to this study could be to incorporate impact analysis to account for changes in the classification tree models when selecting a test case for regression

testing. In our context this means identifying which partitions are affected by the change, and limit the testing to those partitions exclusively, or give them higher priority. Such an extension would not change the results reported in this study, since we would still use the same techniques within each partition. It is, however, an activity that would come with an additional cost, and whether the cost of performing an impact analysis would be worth the gain in improving regression test selection remains to be investigated. On the other hand, it would most likely improve scalability since it would allow us to focus on a subset of partitions.

7 References

[1] Jun-ichi Aoe. *Computer algorithms: String pattern matching strategies*. Practitioners Series. IEEE Computer Society Press, 1994.

[2] Andrea Arcuri and Lionel Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In: *Software Engineering (ICSE), 2011 33rd International Conference on*. 2011, pp. 1–10.

[3] Charles H. Bennett et al. "Information Distance." In: *IEEE Transactions on Information Theory* 44.4 (1998), pp. 1407–1423.

[4] Rudi Cilibrasi, Paul Vitányi, and Ronald Wolf. "Algorithmic clustering of music." In: *Computer Music Journal* 28 (2004), pp. 49–67.

[5] Rudi Langston Cilibrasi. "Statistical Inference Through Data Compression." PhD thesis. Plantage Muidergracht 24, 1018 TV, Amsterdam Holland: Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2007.

[6] Michel M. Deza and Elena Deza. *Encyclopedia of Distances*. Springer, Aug. 2009.

[7] Stefan Droste, Thomas Jansen, and Ingo Wegener. "On the analysis of the (1+ 1) evolutionary algorithm." In: *Theoretical Computer Science* 276.1-2 (2002), pp. 51–81.

[8] Francisco G. Oliveira Neto Emanuela G. Cartaxo Patrícia D. L. Machado. "On the use of a similarity function for test case selection in the context of model-based testing." In: *Software Testing, Verification and Reliability* 21 (2 2011), pp. 75–100.

[9] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. "Achieving Scalable Model-based Testing Through Test Case Diversity." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.1 (Mar. 2013), 6:1–6:42.

[10] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. "Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection." In: *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 327–336.

[11] Hadi Hemmati and Lionel Briand. "An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection." In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. ISSRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 141–150.

[12] Eugene F. Krause. *Taxicab geometry: An adventure in non-Euclidean geometry*. Dover Publications Inc., 1988.

[13] Eckard Lehmann and Joachim Wegener. "Test Case Design by Means of the CTE-XL." In: *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*. 2000.

[14] Dimitris Liparas, Lefteris Angelis, and Robert Feldt. "Applying the Mahalanobis-Taguchi strategy for software defect diagnosis." In: *Automated Software Engineering Journal* 19 (2 2012), pp. 141–165.

[15] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L. Massart. "The Mahalanobis distance." In: *Chemometrics and Intelligent Laboratory Systems* 50.1 (2000), pp. 1–18.

[16] John D. Musa. *Software reliability engineering*. 2nd ed. AuthorHouse, 2004.

[17] Thomas Joseph Ostrand and Marc J. Balcer. "The category-partition method for specifying and generating fuctional tests." In: *Communications of the ACM* 31.6 (1988), pp. 676–686.

[18] Erik Rogstad et al. "Industrial Experiences with Automated Regression Testing of a Legacy Database Application." In: *27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 362–371.

[19] András Vargha and Harold D. Delaney. "A critique and improvement of the CL common language effect size statistics of McGraw and Wong." In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[20] Shin Yoo and Mark Harman. "Regression Testing Minimisation, Selection and Prioritisation: A Survey." In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

Paper 3

Clustering Deviations for Black Box Regression Testing of Database Applications

Author: Erik Rogstad and Lionel Briand

Abstract

Regression tests often result in many deviations (differences between two system versions), either due to changes or regression faults. For the tester to analyze such deviations efficiently, it would be helpful to accurately group them, such that each group contains deviations representing one unique change or regression fault.

Because it is unlikely that a general solution to the above problem can be found, we focus our work on a common type of software system: database applications. We investigate the use of clustering, based on database manipulations and test specifications (from test models), to group regression test deviations according to the faults or changes causing them. We also propose assessment criteria based on the concept of entropy to compare alternative clustering strategies.

To validate our approach, we ran a large scale industrial case study, and our results show that our clustering approach can indeed serve as an accurate strategy for grouping regression test deviations. Among the four test campaigns assessed, deviations were clustered perfectly for two of them, while for the other two, the clusters were all homogenous. Our analysis suggests that this approach can significantly reduce the effort spent by testers in analyzing regression test deviations, increase their level of confidence, and therefore make regression testing more scalable.

Keywords – Software regression testing, test analysis, clustering, regression test deviations.

Abbreviations and Acronyms

SOFIE	The Norwegian Tax Accounting System, the case study system
DART	DAtabase Regression Testing, a tool developed for regression testing of database applications
CTE-XL	Classification Tree Editor - eXtended Logics, a commercial tool for classification tree modeling.
SQL	Structured Query Language
PL/SQL	Procedural Language or Structured Query Language
DML	Data Manipulation Language
EM	Expectation Maximization, an algorithm used for clustering in our case
LSA	Latent Semantic Analysis
SIR	Software-artifact Infrastructure Repository

Notations

$\mathbf{D}$	The set of deviations
$\mathbf{C}$	The set of clusters
$\mathbf{c_j}$	A cluster j in C
$\mathbf{d_i}$	Deviation type i in D
$\mathbf{d_{ij}}$	Deviation type i in cluster j
$\mathbf{E_D}$	Deviation entropy
$\mathbf{E_C}$	Cluster entropy

1 Introduction

Regression testing is a highly important but time consuming activity [12]. A great deal of work has been performed on devising and evaluating techniques for selecting, minimizing, and prioritizing regression test cases [35]. Such techniques are necessary, but unfortunately not sufficient to help scale regression testing to large, complex systems. Indeed, in practice, even with efficient prioritization or selection, numerous regression test deviations may need to be analyzed to determine if they are due to a regression fault or simply the effect of a change. A problem that has been largely ignored so far, but which is highly important in practice, is how to cope with the many discrepancies (deviations) that can be observed when running regression test cases on a new version of a system. In other words, how can we help testers analyze such deviations, which can be due to either changes or regression faults, and decide what are their actual causes. While this is an important problem for regression testing of all types of systems, the types of output from regression tests and

their deviations are logically dependent on the type of system being tested. Thus, we do not believe there is a general solution to assist in the analysis of regression test deviations for all types of systems. In this case study, we focus on database applications, where the regression test output consists of deviations in database manipulations. Although our proposed solution may not be limited to database applications, this is the context where we have evaluated it, and where the reported results are most likely to generalize and be accurate.

In the context of regression testing of database applications, we have proposed a black-box regression test methodology and tool [24]. We model the input domain of the system under test as classification trees, using the tool Classification Tree Editor - eXtended Logics (CTE-XL). Our goal is to identify patterns among deviations by making use of clustering algorithms, the ultimate objective being to identify groups of deviations caused by identical changes or regression faults. As elaborated in Section 5 when investigating the research literature within the context of testing, clustering has most widely been adopted in test case optimization techniques, such as test case selection and prioritization, while the problem we address has been little explored. We perform a thorough assessment of which type of data leads to the most accurate clustering results, using information collected about database manipulations and test case properties. One important contribution of this paper is to assess the accuracy of clustering for grouping regression test deviations in an industrial context, with real changes and regression faults, and evaluate the practical impact for the tester. Our regression test approach targets database applications in general, because it is based on information that should be easy to collect for most systems falling into that category.

The results show that clustering indeed can serve as an accurate strategy for grouping regression test deviations according to their cause, being changes or regression faults. For two out of the four test campaigns evaluated in the study, we achieved perfect clustering results with the clustering strategy that fared best, meaning that there were as many clusters as distinct changes and regression faults, and each cluster only contained one type of deviation (homogenous). For the other two test campaigns, all clusters were still homogenous, but the numbers of clusters were slightly larger than the numbers of distinct deviations, i.e. deviations caused by the same fault or change were spread across more than one homogeneous cluster. That level of accuracy, when grouping deviations, is expected to significantly reduce the test analysis effort, and thus improves the scalability of any regression testing strategy.

The remainder of the paper is organized as follows. Section 2 describes the challenges addressed by this paper, and the industrial context of our study, which in many ways are representative of environments where large database applications are being devel-

oped. Section 3 outlines our proposed solution and the specifics of the clustering strategy, whereas Section 4 describes the industrial case study and reports empirical results. Related works are presented in Section 5, and the paper is concluded in Section 6.

2 Context and background

The main purpose of this section is to provide information about the context in which the problem was identified and defined. This problem, however, is of a general nature, and likely to be present in most database applications. Providing a concrete description of the background of our industrial case study will help the reader understand the scope, importance, and generality of the problem.

Petersen and Wohlin [19] proposed a checklist for context documentation in industrial software engineering. We cover elements of the product facet, the process facet, and the practices and tools and techniques facet in this section. As the subject system is a customized application for the Norwegian Tax Department, we do not regard the organization and market facet relevant in this case, and will not elaborate on that any further. The details of how the case study was performed and the people involved is described in Section 4.2.

2.1 Setting

The Norwegian Tax Department maintains the Norwegian tax accounting system (SOFIE), a system whose main purpose is to collect tax from all taxpayers in Norway. The system was developed as a customized application for the Norwegian Tax Department during the late 2000s, and entered its maintenance phase in 2011. SOFIE serves the daily operation of more than 3,000 end users (i.e. taxation officers), and handles annual tax revenues of approximately 600 billion Norwegian Kroner. It is therefore important to preserve system quality upon changes and new releases, to avoid additional effort to end users and expensive mistakes both for the taxpayers and administration. The system is a very large database application, built on Oracle database technology, developed with standard SQL and PL/SQL programming, along with Oracle Forms user interfaces. In terms of size, the system has more than 1,000 tables in its main database schema, and contains more than 700 PL/SQL packages, and a variety of other database application artifacts (triggers, functions, procedures, views, etc.), in total constituting more than 2.5 million lines of code. The core of the system is batch driven, with periodically scheduled batch jobs processing centralized tasks for all Norwegian taxpayers, whereas the end users access the application through a web interface. The batch jobs process large amounts of data, and their complexity stems from the wide range and diversity of possible input data to

process. Though it is vital for the tax department to avoid releasing defects into the core of the system, the system is complex and difficult to evolve to respond to changes in the tax laws, and is thus prone to regression faults. Therefore, an appropriate regression test procedure is required to guarantee the quality of SOFIE. The releases of SOFIE follow a typical waterfall development process, and the regression testing referred to here is mainly targeted towards system level testing.

2.2 Regression test procedure

Because manual regression testing is not deemed to scale for large database applications, we proposed a partially automated regression test procedure tailored to database applications [24]. It compares executions of a changed version of the program against the original version of the program, and identifies deviations, which are differences in the way the database is manipulated between the two executions. In each test execution, the database manipulations are logged according to a specification by the tester indicating the tables and columns to monitor (hereby called *test configuration*). More specifically, we log Data Manipulation Language (DML) statements, namely inserts, updates, and deletes (whether they are prepared or parameterized statements or not) per test case. These database manipulations are captured using dynamically generated database triggers, based on the test configuration, during the execution of the test. As detailed in Rogstad et al. [24], the database manipulations from each execution are logged in a structured way, and are compared across system versions to produce a set of deviations, indicating either a correct change or a regression fault. The comparison is performed using SQL set operations (minus and union) to capture the differences in database manipulations between two test runs, grouped per test case. In the context of SOFIE and any tax system in general, a test case matches a taxpayer, and all logged data are grouped per taxpayer. The output of regression testing will therefore indicate whether or not there was a deviation between two system versions in the tax calculation for a particular taxpayer. Additionally, to make test selection more systematic and cost effective, following common industry practice, we model the input domain as a classification tree, like the example shown in Fig. 3.1. All relevant properties of the system are modeled as classifications (e.g., Property B: Nr of X), and split into equivalence classes (e.g., 1 through 4, and 5 through 10) representing the range of possibilities in the inputs of the system. For the sake of generality, as other input modeling techniques could have been used, we hereby refer to the classifications in the model as *model properties*, and the values of the equivalence classes (leaf classes in the model) as *model property values*. Based on the model, we generate input partitions (combinations of equivalence classes) according to a given coverage criteria (i.e. pair-wise, three-wise, and all combinations), and each partition represents an abstract test case.

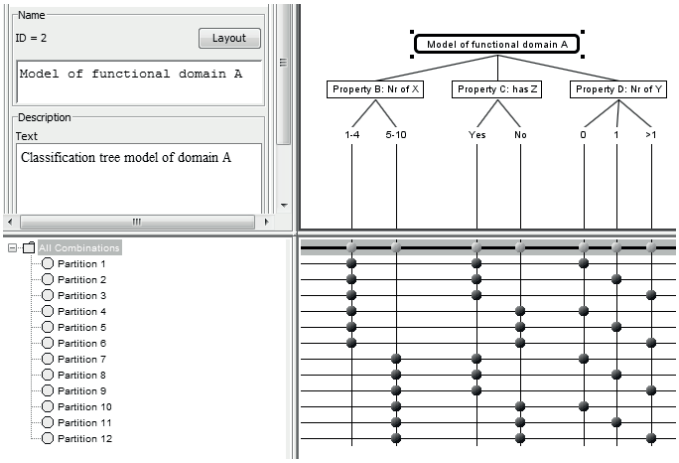


Figure 3.1: An example of a classification tree model in CTE-XL, and the generated partitions (combinations of equivalence classes) that form abstract test cases.

Such a test procedure has the benefit of pinpointing exactly which test case results have changed between system versions. As an example, let us consider a batch processing 10,000 tax calculations from which 100 deviations are identified. We can conclude that no regression faults were visible in the 9,900 test cases with no deviations, while our attention can be directed towards the 100 tax calculations with deviations. Though automated deviation analysis helps a great deal, the tester still has to inspect deviations to determine their cause, and this is a manual, time consuming activity. The number of deviations typically varies with the size of the test suite and the scope of the changes in the particular release under test. But in most cases, the number of deviations is expected to be large, and will require prioritization given the usual project time constraints. Throughout previous SOFIE releases, we have experienced a varying number of deviations across regression test campaigns. This experience is shown in Table 3.1, which shows the number of test cases, the total number of deviations, and the number of distinct groups of deviations with the same cause for various test campaigns.

2.3 Problem formulation

The number of deviations vary from 8 to 522, and though eight deviations are clearly manageable for a tester to manually inspect, in most campaigns there are many more. Imagine spending 15 minutes on average inspecting each deviation, which is a very conservative number in our experience; then analyzing 522 deviations would require 130 hours. Though available time for a tester to carry out a test campaign is a function of its size and importance, spending 130 hours on inspecting deviations is rarely a possibility. We

need a solution to dramatically reduce the time spent in such inspections. By introducing systematic methods for selecting test cases for execution [23], we have been able to reduce the size of the test suites while retaining a good coverage of test scenarios. However, the number of deviations resulting from regression test campaigns is still, in most cases, much larger than what a tester can realistically handle. A trade-off is therefore required, and the tester needs assistance to determine which subset of deviations to focus on to identify as many different regression faults as possible.

Any deviation between the original system output or observable behavior and that of the modified version is seen as a potential regression fault. When presented with the list of deviations, the tester needs support to systematically process them, as relying on experience alone is rather uncertain in such a context where a large number of deviations, mostly different from what was observed in previous versions, have to be analyzed. Without any automated decision support, what we observed is that the current practice for a tester is basically to start at the top of the arbitrary ordered list of deviations and work through as far as time budget permits. But if there is not enough time to inspect all of them, the tester is then left with great uncertainty. Our experience shows that there always is redundancy among the deviations in the sense that several of them are caused by the same change or fault. As shown in Table 3.1, the number of deviations is always larger than the number of distinct deviations, that is the number of groups of deviations related to distinct changes or faults. For a tester to avoid spending unnecessary effort on analysis, we need a systematic way to group the deviations, so that ideally each group matches one distinct deviation only. Then the tester would only have to inspect one deviation from each group to complete the analysis. Given a perfect grouping strategy, the tester would be left inspecting 15 deviations as opposed to 522, or 2 as opposed to 84, to take a couple of examples from the data shown in Table 3.1. The potential savings are therefore highly significant.

Thus, the problem we try to address is to find a strategy to accurately group deviations resulting from the same change or regression fault, in order for the tester to inspect as few deviations as possible while still remaining confident of finding all regression faults.

3 Clustering regression test deviations

Recall that, in our context, a deviation points out a difference between the original program and the changed version of the program for a particular test case. For each test case resulting in a deviation, a detailed list of the specifics of the deviation is provided by our tool. That is, a deviation for a particular test case will indicate the concrete database fields that deviate between the two system versions. Table 3.2 gives a concrete example

Table 3.1: A subset of historic data on the number of deviations produced from regression tests

Test campaign	# Test cases	# Deviations	# Distinct deviations
1	711	33	7
2	3144	182	11
3	5670	522	15
4	1570	8	2
5	94	48	2
6	560	47	2
7	560	84	2
8	151	43	3

Table 3.2: Example of the output of a regression test, i.e. the details of the deviations

Test case	Table	Column	Old value	New value	Program version
1	A	X	5,000	0	Baseline
1	A	X	5,000	4,000	Delta
2	A	X	Inserted	2,500	Baseline
2	A	Y	Inserted	TAX	Baseline
3	A	X	350	Deleted	Delta
3	A	Y	INTEREST	Deleted	Delta

of how the details of the deviations may look like in a trivial case. In this case, three test cases (1, 2, and 3) resulted in a deviation. In test case 1, column X in database table A was updated from 5,000 to 0 in the baseline run of the program, whereas it was updated to 4,000 in the delta run. For test case 2, a record was inserted in database table A with the values 2,500 for column X, and TAX for column Y in the baseline run, but not in the delta run. For test case 3, a record was deleted containing the values 350 in column X, and INTEREST in column Y in the delta run, but not in the baseline run. Given a large number of deviations, each composed of a great deal of detailed information, the amount of information to process is large. In a typical regression test, there may be thousands of rows of deviation details. As a result, recognizing patterns across test cases, indicating similarities among deviations, is a highly difficult task.

We have chosen to base our solution for grouping deviations on clustering, that is algorithms which aim at discovering groups and structures in data in such a way as to maximize the distance between groups, and minimize the distance within groups [32]. In our specific context, we would like to identify a clustering approach that could help us cluster the deviations as accurately as possible, the goals being to have 1) all deviations resulting from the same regression fault or change within the same cluster, and even more importantly, 2) homogenous clusters containing only deviations related to the same fault or change. In other words, the ideal output would have as many clusters as there are

regression faults and changes, and each cluster would only contain deviations matching a unique fault or change.

The most important decision to make is to decide what data to feed into the *clustering algorithm*. As mentioned in Section 2, we model the input domain of the system under test as a classification tree, from which we generate abstract test cases. In addition, during the execution of the system under test, the regression test tool automatically captures all database manipulations executed by each test case in the format *<table, column, operation, old value, new value>*. Thus, the information available as input for clustering is twofold: 1) which model property values each test case covers; and 2) the deviation output describing the specifics of the deviation at a table, column, operation, and value level. It is difficult to tell, a priori, which type of input data will lead to the most accurate grouping. Assessing different input sources and their combinations is therefore an objective of our empirical investigation in the next section.

The input data used for clustering is encoded in a binary matrix, as shown in the upper part of Fig. 3.2. The matrix contains one row per deviation, and for each deviation we indicate with a Boolean value whether a particular table, column, operation, or model property value characterizes the deviation. For example, $table_{1..a}$ denotes each potential table in the deviation output (retrieved from the test configuration), which will be marked 1 (true) or 0 (false) depending on whether or not the table is present in the deviation output for a particular deviation. Similarly, $column_{1..b}$ denotes all monitored database table columns, and $Operation_{1..c}$ denotes the type of database operation that caused the deviation, which is either an insert, update, or delete. This structure is applied at a table level, stating whether the deviation was due to an insert, update, or delete applied to a table. *Model property value*$_1..d$ denotes whether or not the test case causing the deviation covers a certain model property value or not, where a model property value is a leaf class from the classification tree model (e.g., 1 through 4 for Property B in Fig. 3.1).

To exemplify, let us generate an input matrix to the clustering based on the example model shown in Fig. 3.1, and the deviation output shown in Table 3.2. Let *Partition 1* from the example model be the test case specification of *Test case 1* in the deviation output. Similarly, *Partition 2* specifies *Test case 2*, and *Partition 3* specifies *Test case 3*. In this small example, the only entries in the test configuration (tables and columns to monitor) is column X and Y in database table A. Based on the deviation output given in Table 3.2, and the model property values given by Fig. 3.1, the encoded binary matrix would then look like the one shown in Table 3.3. The data from the binary matrix, either representing one single input source (tables, columns, database operations, or model property values) or their combinations, will be used as input to the clustering algorithm.

Once test cases are executed, and deviations are clustered based on the data presented

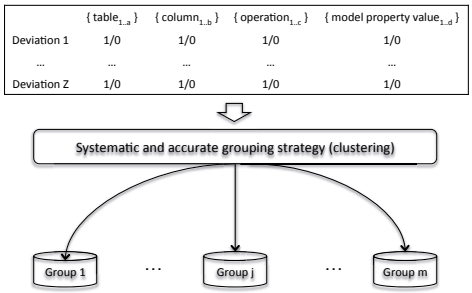


Figure 3.2: The grouping strategy; encode available information and input to a clustering algorithm to group deviations

Table 3.3: A small, artificial example of a binary matrix used as input to clustering.

Deviation	Table	Column		Operation			Model Property Values						
	A	A.X	A.Y	A.INSERT	A.UPDATE	A.DELETE	B.1-4	B.5-10	C.YES	C.NO	D.0	D.1	D.>1
1	1	1	0	0	1	0	1	0	1	0	1	0	0
2	1	1	1	1	0	0	1	0	1	0	0	1	0
3	1	1	1	0	0	1	1	0	1	0	0	0	1

above, we rely on a deviation analysis strategy that is aimed at minimizing the number of deviations to analyze, while covering all regression faults or changes. It is assumed that the regression test analyst will analyze, in turn, at least one deviation from each deviation cluster, in a random order. If each cluster captures deviations corresponding to a unique cause, i.e. change or regression fault, this result will satisfy our objectives. Specific accuracy measures for our clustering algorithm will be discussed in the next section.

Many clustering techniques require the number of clusters to be determined beforehand. However, in our case, we do not know the target number of clusters up front, as the number of distinct deviations will vary across regression test campaigns. Thus, an important requirement when choosing a cluster algorithm to serve our purpose is to select one that does not require a predefined number of clusters as input. However, we would like to limit the maximum number of clusters, as historical data show that the number of distinct deviations, i.e. groups of deviations corresponding to distinct causes, typically represent 2% to 7% of the total number of deviations. Being able to set an upper bound for the number of clusters to approximately 10% of the number of deviations would prevent us from having an unrealistically large number of clusters. Because a part of the deviation analysis strategy is to inspect at least one deviation from each cluster, we target as few clusters as possible. Clustering, as we defined it, can be applied to any type of test case. But of course, if we were doing a white-box approach, we might use coverage information as well for doing clustering.

For practical reasons, we chose to consider cluster algorithms provided by Weka [32],

as it is well documented, has an active wiki and discussion forum, and provides an open source programmable API which is easy to integrate into our existing solution. Among the clustering algorithms provided by Weka, the probabilistic algorithm Expectation Maximization (EM) is the only one that does not require the number of clusters to be predefined (it determines the number of clusters to create based on cross-validation of the input data), and that does provide the ability to set an upper bound for the number of clusters. Consequently, the EM clustering algorithm was our preferred choice for conducting deviation grouping. Inputs to the algorithm include the data set to cluster, the acceptable error to converge, the maximum number of iterations, and optionally the maximum number of clusters. In the case of this study, we have used 1.0E-6 as the minimum allowable standard deviation for convergence, while the maximum number of iterations is set to 100 [32]. Additionally, as stated above, based on historical data, we set the maximum number of clusters to 10% of the number of deviations. If the maximum number of clusters is not given a priori, EM will resort to cross-validation on the input data to set the number of clusters.

In our context, we have a set of vectors representing the deviations, and a set of clusters capturing groups of deviations with common causes. EM will start with an initial guess for clustering, based on the cross-validation of the input data. Then, it estimates an initial distribution of deviations across clusters, i.e. the probability distribution for each deviation, indicating its probability of belonging to each of the clusters. Through iterations, the algorithm will try new parameters based on the outcome of the previous iteration to maximize the log-likelihood of the deviations across the clusters. The algorithm finishes when the distribution parameters converge, or when reaching the maximum number of iterations. The Weka implementation of EM clustering does not use a distance function, but rather estimates a mixture model for clustering, by taking a naive Bayes approach which models each attribute *s*-independent of the others given the cluster. We refer to McLachlan and Krishnan [16] for further details about the EM clustering algorithm.

4 Case study

This section outlines the case study according to the guidelines given by Runeson and Höst [26], and presents the results of the study.

4.1 Objective and research questions

The objective of the case study is to group regression test deviations in such a way that deviations resulting from the same regression fault or change are likely to be contained within the same group, and so that groups only contain deviations related to the same

fault or change. This grouping is expected to increase the efficiency of the tester when performing regression test analysis as, if the above objectives are reached, only one deviation from each group needs to be inspected, rather than picking deviations at random within a limited time frame or inspecting all of them.

The following research questions have been derived from the overall objective of the case study.

RQ1 Can clustering serve as an automated, accurate grouping strategy for grouping regression test deviations?

RQ2 What kind of input data to the clustering process yields the most accurate grouping of the deviations?

RQ3 How much effort can the tester be expected to save in regression test and deviation analysis, when systematically grouping test deviations prior to analysis, as opposed to the current random inspection?

4.2 The case and data collection

We investigated test deviations from the regression tests of SOFIE, the tax accounting system of the Norwegian Tax Department. Most of the context documentation is given in Section 2, whereas we detail the specifics of how the case study was conducted here.

The test team of SOFIE ran four regression test campaigns using the regression test tool DART [24] on consecutive releases of SOFIE. One of their testers was responsible for the test execution, while one tester (having the necessary domain knowledge) conducted the analysis of the test results. As expected, the specifics of the tests varied across releases, as different parts of the system were affected by the scope of the release. Once the deviations were classified as either regression faults or changes to be verified, we applied our clustering strategy, and evaluated its accuracy and potential efficiency gains for the testers. Details about the particular test campaigns used in the study are provided in Table 3.4, showing the number of test cases executed, the number of deviations resulting from the test, and the number of distinct deviations with the frequency of each distinct deviation in brackets. As for the number of tables and columns monitored during the regression test execution, the order of magnitude for the test campaigns in our case study was 10-12 tables, spanning 63-83 table columns, whereas the size of the classification tree models varied from 26 to 127 model property values.

For the purpose of the case study, we collected data about actual regression tests across various releases of SOFIE. More specifically, information was collected about the test configuration, the model of the test domain, and the list of deviations produced, which

Table 3.4: The subject regression test campaigns for the case study

Test Campaign	# Test cases	# Deviations	# Distinct deviations [Frequency]
1	94	48	2 [41 (fault), 7 (fault)]
2	560	47	2 [19 (fault), 28 (change)]
3	560	84	2 [56 (change), 28 (fault)]
4	151	43	3 [3 (fault), 10 (change), 30 (change)]

were manually analyzed and categorized by the test team as regression faults or changes. According to the definitions given by Runeson and Höst [26], this process corresponds to second degree data collection, as we are directly in control of the collection of our raw data through observations of the regression tests on the subject system.

The SOFIE project was selected as it is large, complex, and representative of database applications developed by the tax department and other public administrations. Furthermore, and it is important when studying regression testing, SOFIE undergoes frequent changes due to general maintenance and changes in taxation laws.

4.3 Evaluation

The accuracy of the clustering algorithm, which is in question in both RQ1 and RQ2, is measured in terms of its entropy. Entropy is a measure of unpredictability [10], and we apply it to evaluate two distinct dimensions: deviation, and cluster.

- Deviation entropy is the measure of the spread across clusters of deviations matching the same fault or change. Ideally all related deviations should only be contained in one cluster.

- Cluster entropy is the measure of the purity of a cluster. Ideally a cluster should only contain deviations matching one specific fault or change.

Entropy gives us a normalized measure of the purity of each cluster, and the spread of each type of deviation across clusters, while additionally giving an overall measure for a set of clusters, enabling us to compare the overall clustering accuracy for different kinds of inputs. A perfect clustering would have as many clusters as there are regression faults and changes, and each cluster would only contain deviations matching a unique fault or change, thus yielding an entropy of zero. The worse the grouping, the higher the entropy level, with the worst case occurring when deviations matching specific faults or changes are equally spread across the clusters. In the remainder of the text, we will use deviation group to mean an actual group of deviations matching a specific fault or change. The formula for measuring deviation entropy (E_D) [10], more specifically the spread of a particular deviation group d_i in D across the m clusters (C), is given by (3.1). For each

of the m clusters c in C, we compute the number of deviations of type i that belong to cluster c_j (d_{ij}) divided by the total number of deviations in group i ($|d_i|$). This ratio is then employed in the usual information theory entropy formula for all clusters and groups. In general, in all our entropy calculations, we have used the natural logarithm ($log_e(p)$), and taken $0log(0)$ to be 0, to be consistent with the limit $\lim_{p\to 0+} plog(p) = 0$, stating that when p converges sufficiently close to zero, $plog(p)$ evaluates to zero.

$$\mathrm{E_D}(d_i, C) = -\sum_{j=1}^{m}(\frac{d_{ij}}{|d_i|})log(\frac{d_{ij}}{|d_i|}) \tag{3.1}$$

The formula for measuring cluster entropy (E_C), namely the purity of a cluster c_j, is given by (3.2). For each of the n deviation groups, we compute the number of occurrences of d_i in c_j (d_{ij}) divided by the total number of deviations in c_j. This ratio is then employed again using the information theory entropy formula for all clusters and groups.

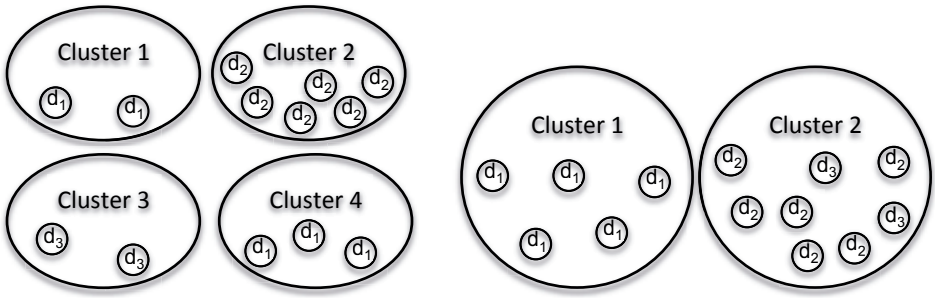
$$\mathrm{E_C}(D, c_j) = -\sum_{i=1}^{n}(\frac{d_{ij}}{|c_j|})log(\frac{d_{ij}}{|c_j|}) \tag{3.2}$$

The formula for total deviation entropy (E_{D_TOT}), and total cluster entropy (E_{C_TOT}) are given in (3.3), and (3.4), respectively. They are basically the sum of (3.1) and (3.2), across all deviation groups, and clusters, respectively.

To demonstrate the calculations in practice, we will calculate the entropy for one deviation and one cluster from the examples given in Fig. 3.3. The calculation of deviation entropy for deviation d_1 in Fig. 3.3(a) is illustrated in (3.5). In total, there are 5 deviations of type d_1, meaning the cardinality of deviation d_1 ($|d_1|$) is 5. The deviation group d_1 is spread across two clusters, c_1 and c_4, containing 2 and 3 deviations of type d_1, respectively. The other two clusters do not contain any deviations from group d_1, so those parts of the equation evaluate to zero. The calculation of cluster entropy for cluster c_2 in Fig. 3.3(b) is illustrated in (3.6). The cardinality of cluster c_2 ($|c_2|$) is 8, and it contains 6 deviations of type d_2, and 2 deviations of type d_3. As cluster c_2 does not contain any deviations from deviation group d_1, the first part of the equation evaluates to zero. All other deviation and cluster entropy calculations for the examples given in Fig. 3.3 would evaluate to zero.

$$\mathrm{E_{D\_TOT}}(D, C) = -\sum_{i=1}^{n}\mathrm{E_D}(d_i, C) \tag{3.3}$$

$$\mathrm{E_{C\_TOT}}(D, C) = -\sum_{j=1}^{m}\mathrm{E_C}(D, c_j) \tag{3.4}$$



(a) Clustering with zero cluster entropy (all clusters are pure). 13 deviations of three different types d_1, d_2, and d_3 spread across four clusters.

(b) Clustering with zero deviation entropy (all deviations of the same type are contained within the same cluster). 13 deviations of three different types d_1, d_2, and d_3 spread across two clusters.

Figure 3.3: Zero cluster entropy versus zero deviation entropy.

$$\mathrm{E_D}(d_1, C) = -\sum_{j=1}^{4}(\frac{d_{1j}}{|d_1|})log(\frac{d_{1j}}{|d_1|}) = (\frac{2}{5})log(\frac{2}{5}) + (\frac{0}{5})log(\frac{0}{5}) + (\frac{0}{5})log(\frac{0}{5}) + (\frac{3}{5})log(\frac{3}{5}) \approx 0.6730 \tag{3.5}$$

$$\mathrm{E_C}(D, c_2) = -\sum_{i=1}^{3}(\frac{d_{i2}}{|c_2|})log(\frac{d_{i2}}{|c_2|}) = (\frac{0}{8})log(\frac{0}{8}) + (\frac{6}{8})log(\frac{6}{8}) + (\frac{2}{8})log(\frac{2}{8}) \approx 0.5623 \tag{3.6}$$

From a practical viewpoint, it is more important to obtain zero cluster entropy than zero deviation entropy. If the cluster entropy evaluates to zero, we are certain that each cluster only matches one regression fault or change. Under such circumstances, the tester would always conduct a complete deviation analysis when inspecting one deviation from each group, only with the potential risk of having to analyze a few more deviations than necessary if the deviation entropy is not perfect. On the contrary, if the cluster entropy significantly differs from zero, then inspecting only one deviation from each group would represent a significant risk. The ideal target is a total entropy level of zero; but if we have to compromise, a zero cluster entropy is preferred over a zero deviation entropy. That is, the situation in Fig. 3.3(a) is preferred over the situation in Fig. 3.3(b).

To evaluate the effort spent by the tester in analyzing regression test deviations for RQ3, we measure the expected number of deviations to be inspected to identify all deviation groups. This approach relies on an assumption regarding how the clustering results are used, and in our context we assume a round robin random selection of deviations from the clusters. That is, the tester would randomly select clusters in a round robin manner,

and within each cluster select a random deviation (under the assumption that they all match the same regression fault or change). For each test campaign, we count the number of deviations to inspect until all deviation groups are covered. We repeat the exercise 1000 times to account for randomness in sampling, and report basic statistic measures such as minimum, median, maximum, average, and standard deviation regarding the number of deviations. Additionally, we conduct non-parametric, two-tailed Mann-Whitney U-Tests [2] to check the statistical significance of the difference in inspection effort when the cluster-based approach was compared against the random inspection currently in use. We also report $\hat{A}_{12}$ effect size measurements to assess the practical significance of the differences. An $\hat{A}_{12}$ effect size measurement value of 0.5 indicates that there is no difference between the two samples compared, while the further away from 0.5, the larger the effect size between the samples. We have categorized the effect size into the standard Small, Medium, and Large categories, which are usually defined as Small < 0.10, $0.10 <$ Medium < 0.17, and Large > 0.17, the value being the distance from the 0.5 threshold [30].

4.4 Results

This section reports our case study results when clustering is applied to group regression test deviations. During the course of the study, four regression test campaigns were executed, from which data were collected. Only a subset of the eight test campaigns given as examples in Table 3.1 were included in the current study as, due to changes in the project environment, not all required data were available for some past releases. Relevant data for the different regression test suites are shown in Table 3.4. For each of them, we went through the following activities.

1. Preparing input data: For each type of input data (tables, columns, database operations, and model property values), we generated an input matrix (as shown in Fig. 3.2) for the *clustering algorithm*. In addition, we generated matrices combining all combinations of the input types.

2. Measuring entropy: For each of the various input matrices provided, we carried out the clustering, as described in Section 3, and measured the entropy level for each deviation group, and each generated cluster, to assess clustering accuracy. It should be mentioned that the execution time of the clustering is negligible (0 to 2 seconds), and is therefore not given any further attention in the study.

3. Assessing expected inspection effort: Based on the most accurate clustering approach overall, we estimated the expected number of deviations that must be inspected by the tester to inspect all distinct deviations. This estimate is compared

Table 3.5: Entropy measurements for each combination of input values, and for each of the four regression test campaigns.

			Entropy							
			Test 1		Test 2		Test 3		Test 4	
			Deviation	Cluster	Deviation	Cluster	Deviation	Cluster	Deviation	Cluster
Cluster input	Single	Tables	0.827	0.000	1.129	0.000	0.000	0.000	0.000	0.000
		Columns	0.920	1.310	0.616	0.474	0.458	0.000	1.295	0.540
		Operations	0.720	0.690	0.410	0.462	0.000	0.000	0.563	0.000
		Model Property Values	0.526	0.685	1.385	0.679	0.000	0.637	1.619	0.855
	Pair	Tables and columns	0.826	0.410	0.693	0.000	0.458	0.000	1.089	0.000
		Tables and operations	0.416	0.000	1.428	0.000	0.000	0.000	0.943	0.000
		Tables and model property values	1.722	0.410	2.277	1.235	0.000	0.000	0.000	0.540
		Columns and operations	**0.416**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**1.089**	**0.000**
		Columns and model property values	0.979	0.690	0.683	0.000	1.040	0.000	1.074	0.540
		Operations and model property values	1.242	1.140	1.779	0.637	0.693	0.000	0.000	0.540
	Three	Tables and columns and operations	0.813	0.000	1.428	0.000	0.000	0.000	1.089	0.000
		Tables and columns and model property values	1.062	0.000	1.684	0.000	0.000	0.000	0.673	0.540
		Tables and operations and model property values	0.494	0.000	0.598	0.000	1.974	0.271	0.991	0.540
		Columns and operations and model property values	1.015	0.689	1.684	0.000	1.890	0.000	0.000	0.540
	All	Tables and columns and operations and model property values	0.416	0.000	1.747	0.683	0.693	0.000	1.089	0.540

against an inspection strategy where deviations are selected at random, because this is the current practice when inspecting the regression test deviations for our subject system, and probably in many other test environments.

Accuracy of clustering

Table 3.5 shows a summary of the *deviation entropy* and the *cluster entropy* per cluster input for each of the subject regression test campaigns. The full list of details, regarding entropy per deviation and cluster, along with the actual distribution of deviations across clusters per input type, and per test, is made available in the Appendix.

The first noteworthy result is that for three of the test campaigns (two, three, and four), a perfect clustering was achieved for at least one type of cluster input, i.e. each row in Table 3.5. By perfect, we mean a total entropy level of zero per test campaign, indicating that there are as many clusters as the number of distinct regression faults and changes, and all clusters are homogenous. A concrete example is the input pair *columns and operations* for test campaign number two. Another interesting point is that four types of inputs result in a zero cluster entropy for all test campaigns, which is our priority. The input types are *tables*, *tables and operations*, *columns and operations*, and *tables and columns and operations*. As stated in Section 4.4, achieving a zero cluster entropy is most important from a practical point of view, as it ensures that all cluster groups are homogeneous, containing instances of one deviation group only. Such a situation allows the tester, in all confidence, to conduct the deviation analysis by only inspecting one deviation from each group. The fact that a total entropy level of zero was obtained for three out of four test campaigns, along with the fact that zero cluster entropy was achieved for all test campaigns for certain input types, shows that we are able to cluster

regression test deviations with a high level of accuracy. Thus, regarding RQ1, we conclude that clustering can serve as an automated, accurate strategy for grouping regression test deviations.

Preferred input data

The best cluster input overall is the combination of *columns and operations*. This strategy resulted in perfect clustering for two of the test campaigns, while also ensuring zero cluster entropy for the other two test campaigns. Using only tables as input yielded similar results, but the overall entropy is nevertheless lower when using the combination of columns and operations. Relying only on tables when clustering would also be more sensitive to the number of tables monitored during the test. If very few tables were monitored, the approach would most likely be too coarse and inaccurate. Thus, relying on the combination of columns and operations seems to be at an appropriate level of granularity. In general, using one source of input seems too limited for the clustering algorithm to yield the best results. For example, using only columns or operations separately provides unsatisfactory results, while combining them leads to very accurate clustering. On the other hand, using all input type combinations or even triples seems unnecessary. With regards to RQ2, we conclude that the combination of *columns and operations* is the input combination yielding the most accurate grouping of the regression test deviations.

Please recall from Section 3 that in our approach we have used a binary representation of our regression test database manipulations and test case properties. Alternatively, it would also have been possible to count the number of occurrences of each table, column, and operation in the deviation output. If we were to venture into even more details, one could also account for the order of operations; however, that would be slightly more intricate to encode into meaningful cluster inputs. Nevertheless, because the results using only the presence of each database element are already so accurate, there was not much room for improvement in our case study. Such alternatives may, however, be worth investigating in other contexts.

One last thing to note is that nearly all clustering results based on model properties yield poor results, indicating they are not suited to cluster deviations. A plausible explanation is that the model describes the input domain of the regression test rather than the output of the test. Although the two are related, the relationship is complex, and the results suggest that the information available in the deviation output is more relevant to clustering the deviations.

Table 3.6: The number of deviations needed to be inspected by the tester to cover all distinct deviations

Test		Inspection strategy	Minimum	Median	Maximum	Average	Std.-dev.
Test Campaign 1	Deviations: 48	Clustered Round Robin	2	2	3	2.324	0.468
	Distinct deviations: 2	Random Inspection	2	5	24	6.564	4.745
Test Campaign 2	Deviations: 47	Clustered Round Robin	2	2	2	2	0.000
	Distinct deviations: 2	Random Inspection	2	3	12	3.111	1.568
Test Campaign 3	Deviations: 84	Clustered Round Robin	2	2	2	2	0.000
	Distinct deviations: 2	Random Inspection	2	3	13	3.337	1.828
Test Campaign 4	Deviations: 43	Clustered Round Robin	3	4	5	4.097	0.831
	Distinct deviations: 3	Random Inspection	3	10	38	12.063	7.451

Inspection effort

Given accurate deviation clusters, we assume here an inspection strategy which involves picking a random cluster in a round robin manner, and then, within each cluster, picking a random deviation. This procedure is continued until all faults and changes are covered, and we measure, as a surrogate measure for effort, the expected number of deviations a tester would need to inspect to cover all deviation groups. This procedure is repeated 1000 times to account for randomness in sampling. The results are shown in Table 3.6, where *Clustered Round Robin* is compared against the *Random Inspection* strategy currently in use in the SOFIE project.

Compared to inspecting all deviations, the effort of the tester is dramatically reduced when clustering deviations and inspecting test results following the round robin procedure stated above. In three out of four regression test campaigns, the median number of deviations to inspect to cover all faults and changes corresponds to the actual number of different faults and changes. In other words, we obtain perfect results. In the worst case, we need to inspect a few more deviations, but never more than the number of formed clusters. And recall from Section 3 that, as part of the clustering strategy, we impose an upper bound for the number of clusters to 10% of the number of deviations. Thus, the number of clusters remains manageable, and the number of deviations to inspect remains very reasonable.

The statistical tests and the $\hat{A}_{12}$ effect size measurements are provided in Table 3.7, where the p-value from the statistical tests are reported, along with the categorized $\hat{A}_{12}$ measures with the actual value given in parentheses. The results of the tests show that the cluster-based inspection is significantly better than random inspection for all test campaigns, and the difference in effect size is large for all campaigns, according to standard evaluation scales. This result is also clearly visible from the differences in median and average values in Table 3.6. Between the cluster-based and random inspections, the relative increase in the number of deviations to inspect is between approximately 50% (test campaigns 2, and 3 with 2 vs. 3 deviations) and 150% (test campaigns 1, and 4 with 2 vs. 5, and 4 vs. 10 deviations, respectively). When addressing RQ3, because we

Table 3.7: Mann-Whitney U-tests, and $\hat{A}_{12}$ effect size measurements when comparing inspection effort across clustered inspection, and random inspection

Test campaign	**Cluster-based (A), Random (B)**		
	P-value	**Superior**	**Effect size**
1	< 0.0001	A	Large (0.817)
2	< 0.0001	A	Large (0.754)
3	< 0.0001	A	Large (0.772)
4	< 0.0001	A	Large (0.888)

expect the difference in effort spent on analyzing deviations to be proportional to these differences, our results suggest significant savings when using clustering.

Nevertheless, random inspection yielded better results than we expected (the median number of inspections is 5, 3, 3, and 10). This result stems from a limited number of deviation groups (distinct deviations) present in our subject regression test campaigns. In test campaigns 2 and 3, both deviation groups present are represented by a large number of deviations, which favor a random strategy. For test campaigns 1 and 4, where there exist deviation groups containing relatively few deviations, and in the latter case also more deviation groups, the random approach shows a significant increase in the average number of deviations to inspect and its standard deviation. Therefore, good results for a random inspection strategy are less likely in releases with a larger number of regression faults and changes, especially in the presence of an uneven distribution of corresponding deviations. A detailed analysis shows that the results from the random approach are always worse than those of the clustering approach, and show a significantly higher standard deviation (ranging from 1.568 to 7.451 as opposed to 0.000 to 0.831), thus resulting in more uncertainty in terms of the number of inspections one may have to go through. The worst maximum number for random inspections is 38, which is a significant portion of the 43 deviations present in the test campaign. In contrast, the cluster-based inspection approach provides far more predictable results, which is important in our context to prevent regression faults from slipping though. From our results, which are of course to be confirmed in other studies, this systematic approach guarantees that at least one deviation is analyzed for each fault or change by inspecting only one deviation from each cluster.

To summarize, in practice when conducting a cluster-based inspection, the tester would analyze one deviation from each cluster (one-per-cluster sampling). And most importantly, given the results we obtained, the tester would do that analysis while remaining confident he or she will cover all deviation groups, which are distinct faults and changes. In our case study, that approach would correspond to the following results for the four test campaigns. The tester would inspect 3 out of 48, 2 out of 47, 2 out of 84, and 5 out of 43 deviations for the four test campaigns, respectively. This inspection would defi-

nitely entail highly significant savings. In contrast, when inspecting an arbitrarily ordered list of deviations, the tester would have to inspect all of them, or as many of them as possible given time constraints, to gain sufficient confidence in the test results. In the latter case, the tester would have no guarantee that he has covered all or most faults and changes. Thus, a cluster-based inspection strategy is in practice far better than a random inspection, and yields significant savings, especially in the context of large regression test suites.

4.5 Threats to Validity

In this section, we discuss the generality and potential threats to validity of the research, following the classification scheme of Yin [34], as further described for software engineering in the guidelines by Runeson and Höst [26].

Construct validity

We needed to measure the accuracy of clustering when applied for grouping regression test deviations. By measuring entropy across two dimensions, namely deviation entropy and cluster entropy, we capture two complementary aspects of accuracy required to investigate RQ1 and RQ2. Other accuracy measures, such as standard recall and precision for classifiers, were also considered to assess our clustering results. But unlike classification, clustering results are not compared to a ground truth, telling us what should be the correct classification. We could of course have measured precision and recall based on the majority of changes and faults in each cluster, but that would have yielded specific values for each cluster, and not overall measurement values for all clusters. That is why we chose to rely on entropy, which gives us, for a clustering algorithm and a set of inputs, a normalized measure of the purity of the clusters, and the spread of the deviation groups across clusters.

The inspection effort, addressed in RQ3, is expected to vary across test campaigns and deviations. Thus, we have used the expected number of deviations to inspect as a surrogate measure of inspection effort, and reported standard descriptive statistics, such as median, average, and standard deviation. Inspection effort, on average and over many deviations, is expected to be proportional to the number of inspected deviations.

Internal validity

To assess the accuracy of deviation clustering, we relied on the manual inspection and categorization of the regression test deviations performed by software engineers. The resulting categories of regression test deviations served as a baseline against which we

compared deviation clusters. Ideally, more than one person should have taken part in this process to avoid mistakes. However, because of the cost associated with the task of inspecting all the deviations from each test campaign in a real development environment, this task was performed by one engineer only.

The use of classification trees to specify and automate test cases is not a limitation of the applicability of our approach as clustering results clearly show that model property values, the only information specific to classification trees, are not a useful clustering input. Second, the use of classification trees for black-box testing is widespread and similar to other black-box test specification techniques.

External validity

The fact that the study only involves one system is a potential threat to the external validity of the study. However, the focus on a representative database application, using standard and widely used database technology, combined with real regression test campaigns and faults, helps mitigate this threat. Ideally, one would always want to run multiple studies to better ensure the generalizability of the results. However, to replicate the study, we need actual regression test suites, for actual versions of the system, with known and actual regression faults diagnosed by engineers. Then we need, for each version, to re-run all test cases, and monitor the manipulations on the database, the latter being properly populated with data. That type of data is just not available for open source systems, and to replicate our study on another real world, large scale system would be a major undertaking requiring the collaboration of another development organization.

Regarding the representativeness of our subject system, recall from Section 2 that the subject system is built on standard Oracle database technology. As Oracle is the world market leader both in terms of applications platforms and database management systems [9], the subject system is built on technology that is widespread across the world. Moreover, the chosen subject system is very similar to many other such database applications developed by the Norwegian government and other administrations, with a typically long lifespan. These systems process large amounts of data, leading to a wide variety of possible test scenarios, which makes testing challenging, and the need for assistance crucial in the analysis and diagnosis of regression tests. We have also evaluated several test campaigns, spanning different test models and different functional domains of the system under test, to ensure a certain variety within our case study. The information used as input to clustering should also generalize in the context of database applications, as this kind of information should be easy to collect from any system falling into that category. The database information is also captured through a mechanism separate from the code running on top of the database, i.e. whether the manipulations are performed by

a batch, a procedure, or a graphical user interface does not matter. Captured will be all manipulations performed by any code executed by a test case in the executed test suite on the table columns monitored during test execution.

Borg and Runeson [3] showed a significant dependency between the results obtained and the data characteristics of the studied case in the context of information retrieval in software engineering. Transferred to our case, this could mean that the calibration of the clustering method, that is selecting which input data provides the best clustering output, may depend on the context. If so, it would require a full deviation analysis to calibrate the method in a new context. Then, the gain in inspection effort would not materialize until the method is satisfactorily calibrated, which would be at best after the first test run. Note, however, that the best clustering approach in our case study showed stable, consistent results across the test campaigns. Nevertheless, over time, the need to recalibrate the method at certain intervals may arise. But given a system with a long lifespan, numerous regression tests would be required throughout its existence, thus making the benefits of clustering deviations significant across many releases.

Conclusion Validity

In Section 4.4, we executed the deviation sampling 1000 times per test campaign and per sampling strategy to account for randomness. We used a non-parametric statistical test to compare statistically independent samples, corresponding to different strategies, that make no assumption about data distributions. Thus, we have sufficient statistical power, and use appropriate tests to draw statistical conclusions.

5 Related work

This case study is about supporting the analysis of regression test discrepancies by clustering them to guide their inspection. Thus, we have chosen to place our work in the context of clustering related to testing and fault categorization. We refer to Rogstad et al. [24] for works related to regression testing of database applications, which are not directly relevant to what is presented in this paper as we do not make any assumption about how test cases are selected.

In the context of testing, clustering has been most widely used for test optimization purposes, such as test case selection and test case prioritization. Test case selection aims at selecting a subset of all test cases in a test suite for execution, while test case prioritization allocates a priority to each test case resulting in a test suite of ordered test cases. Both techniques usually try to maximize fault detection, either in the selected subset or in the high priority test cases. Early clustering work in relation to these topics emerged in the

last 20 years, but most particularly surged in the last 4 to 5 years. Although most of the papers are focused on test cases, there are also a few papers applying clustering to fault classification. The following subsections summarize the main contributions of clustering in each application area. We will remain brief on the papers related to test optimization techniques, while going more in depth for the ones on failure clustering, as those are more closely related to our work.

5.1 Clustering-based test case selection

Dickinson et al. [7] evaluated the effectiveness of cluster analysis for finding failures, by building on the work of Podgurski et al. [20, 22] on what was labeled as *Cluster Filtering*. Cluster filtering is the combination of a clustering metric and a sample strategy used to filter executions (test cases) to select those exhibiting unusual behavior, hypothesizing this approach would lead to early fault detection. They combine varying inputs to the cluster algorithm with different sampling strategies to conclude that cluster filtering is more effective (detects more faults) than simple random sampling, the best approach being adaptive sampling combined with input data giving extra weight to unusual profile features.

To select a subset of test cases for regression testing, Parsa et al. [18] cluster test cases based on their execution profile, and then sample test cases until the code coverage level of the original test suite is satisfied. Compared to the well known H-algorithm [11], their approach leads to improved fault detection rates for similar sized test suites, or greater test suite reduction for similar fault detection rates.

Focusing on the elimination of duplicate test cases, Vangala et al. [29] address test case minimization in their clustering-based selection approach. They use the program profile, including code coverage, the number of times a block or arc was executed, and static analysis of the source code as a basis for clustering, using thresholds for redundancy detection. On average, they identified 10-20% redundant test cases with a 70% accuracy.

Zhang et al. [37] proposed a regression test selection technique by clustering execution profiles of modification-traversing test cases. Compared to the safe selection technique proposed by Rothermel et al. [25], their approach produced smaller test suites, retaining most fault-revealing test cases. Chen et al. [6] extended the work by Zhang et al., and introduced semi-supervised clustering into their regression test selection approach. They used two pair-wise constraints between test cases, namely must-be-in-same-cluster, and cannot-be-in-same-cluster, to guide the clustering process. The constraints are derived from the previous test results, and used to state whether pairs of test cases must be in the same cluster, or not. The results showed improved fault detection rates up to 30% when compared to unsupervised clustering.

As opposed to the above-mentioned works, which all are white-box clustering approaches based on execution profiles at the source code level, Sapna et al. [27] applied clustering for specification-based testing. To achieve effective test case selection, they clustered test scenarios from UML activity diagrams to select dissimilar test cases. While the results showed improved fault detection rates over random selection, results did not lead to the desired property of being able to detect most faults with small numbers of test cases.

In the context of Web Applications, Liu et al. [15] collected user-sessions from the application, which in turn were used as test cases. They clustered user-sessions to select them for testing. With the reduced set of test cases, they achieved nearly the same fault detection rate and code coverage level as with the entire test suite.

Wang et al. [31] investigated test case selection in observation-based testing. They also clustered test cases based on their execution profile, but then assigned weights to the clustered test cases based on a ranking metric. They calculated a score for every function covered by the test case based on complexity, where functions receiving high scores are considered more likely to contain bugs. Compared to an execution-spectra-based sampling strategy [33], the weighted approach showed improved fault detection rates with reduced test suite sizes.

To summarize the cluster-based approaches to selection, most of them 1) take a white-box approach by using information from execution profiles at a source code level as a basis for the clustering, Sapna et al. being the exception with a black-box strategy based on activity diagrams; 2) use Agglomerative Hierarchical Clustering or k-means clustering; and 3) are evaluated on one or more open source programs.

5.2 Clustering-based test case prioritization

To benefit from human expert knowledge in test case prioritization, Yoo et al. [36] employ clustering to scale the use of manual pair-wise comparisons of test cases to large test suites. By clustering the test cases hierarchically, the workload of pair-wise comparisons is divided into smaller, more manageable tasks, called intra-cluster and inter-cluster prioritization. Compared to classic statement-coverage based ordering, their clustering-based prioritization fared better for 9 out of 16 test suites, with an average improvement in the fault detection rate of 6.5%.

Simons et al. [28] used clustering for the purpose of regression test case prioritization. Test cases are clustered based on characteristics from the test profiles and execution behavior, followed by an initial random one-per-cluster sampling. If they come across a failed test case (based on information from the previous test), the k nearest neighbors from that test case are selected from within the cluster. Their approach labeled *adaptive*

Table 3.8: Overview of studies related to failure clustering

Reference	Task	Input information	Context	Subject programs
[21]	Cluster failure reports	Automatically generated failure reports	White-box testing	3 open source programs (compilers)
[13]	Cluster failed test cases	Test case execution trace	White-box GUI testing	A messenger application
[8]	Cluster failed test cases	Terms from the executed source code	White-box testing	1 open source program from SIR

failure pursuit sampling was compared to original failure pursuit sampling, yielding a significant increase in the average fault detection rate.

Contrasting with most existing works, Carlson et al. [5] conducted an industrial case study on the use of clustering to improve test case prioritization. They cluster test cases based on code coverage and test case information from the version control system, and then prioritize test cases within and between clusters by using software metrics such as code complexity, fault history, and code coverage. Prioritization incorporating clustering did on average provide a 40% improvement in the fault detection rate over prioritization without clustering.

Arafeen et al. [1] investigate whether requirements information, when added to traditional code analysis information, would improve cluster-based test case prioritization. They use text mining to cluster test cases based on requirement similarities and code complexity information, combined with different sampling strategies. Their results indicate that the use of requirement information during test case prioritization can be beneficial, showing an average improvement in the fault detection rate of approximately 30% compared to code metric prioritization.

To summarize, the clustered-based approaches to prioritization all 1) take a white-box approach by using information from execution profiles at the source code level as a basis for the clustering, 2) use Agglomerative Hierarchical Clustering or k-means clustering, and 3) are evaluated on one or more open source programs from SIR, Carlson et al. being the exception reporting on an industrial case study.

5.3 Clustering for failures classification

Whereas the works mentioned in the previous sections focus on test case clustering, the works presented in this section address the classification of failures using clustering to work through them more efficiently. Therefore, the work presented in this section is the most similar to our work, as the focus is to find out which failures to investigate.

Many software products today have the ability to detect some of their own runtime failures, and automatically generate failure reports to facilitate debugging. Whereas automated failure reporting is a significant asset, it is likely to produce more reports than the developers can effectively handle. Podgurski et al. [21] use clustering to group failure reports which exhibit similar execution profiles with respect to a set of selected features.

They used the Clara clustering algorithm from S-PLUS based on the k-medoids clustering criterion [14], while utilizing the Calinski-Harabasz index [4] to determine the number of clusters. When evaluated on three open source subjects (all compilers), they found that, in most of the clusters created, a majority of the failures were due to the same cause. Overall, groups of failures with the same cause tended to form fairly cohesive clusters. However, a few large, non-homogeneous clusters were created containing sub-clusters consisting of failures with the same cause.

When a large number of failed test cases are reported from a test automation system, recommending some representative test cases as a starting point for debugging can be helpful to developers. In the context of graphical user interface testing, Chien-Hsin et al. [13] proposed clustering as a solution to provide such help. More concretely, they collected test case execution traces through instrumented code, calculated a similarity matrix according to an adapted version of the Needleman-Wunsch Algorithm [17], and clustered the failed test cases with Agglomerative Hierarchical Clustering. The developer was then expected to fix failed test cases following a one-per-cluster sampling strategy. A case study on a messenger application (55,000 lines of code) with seeded faults indicated reduced bug fixing effort when following the proposed strategy.

While many of the clustering-based approaches in testing rely on control-flow analysis, mostly considering the execution profiles of the test cases, DiGiuseppe et al. [8] suggest utilizing latent-semantic analysis (LSA) to capture the semantic intent of an execution. They use LSA in a concept-based execution clustering technique to categorize executions that fail due to the same fault, similar to what Arafeen et al. [1] subsequently proposed. The main difference with the latter is that it is applied at a code execution level rather than based on requirements information. They instrument and execute the program, parse the source code to identify words, compute the weighted term frequency and inverse document frequency for each term, and then cluster the test cases using agglomerative hierarchical clustering. When evaluated on one open source program from the Software-artifact Infrastructure Repository (SIR), concept-based clustering was able to achieve similar cluster purity (94%), but substantially reduced the number of clusters (by 70%) when compared to pure control-flow-based clustering. Indeed, this technique was able to detect similar executions despite differing control paths being executed.

A summary of the works most closely related to ours is provided in Table 3.8. To summarize, in contrast to those works, the current paper is an industrial case study focusing on clustering regression test deviations based on test case specifications and runtime database changes in the context of database applications. No existing work has addressed that important problem. In other words, our work differs with respect to the task supported, the type of information used as a basis for clustering, the types of test

cases, and the application context.

6 Conclusion

One main challenge in large scale regression testing is to analyze observed deviations on new software versions to decide whether they are symptoms of regression faults or the logical result of changes. In a specific context, we have investigated the use of clustering to group regression test deviations to assist and prioritize their analysis. Our focus is on the functional, black-box regression testing of database applications, which are widespread in many application domains, and usually require substantial regression testing over a long lifespan. We compare runtime database manipulations between executions of the original and changed versions of the system under test. We collect information such as the tables and columns subject to manipulation, along with the old and new values of the fields. Note that such information should be general, and easy to collect for any database application. The output of the regression test is the set of test case deviations, along with the specific details of the deviations. The analysis of past regression test campaigns shows that many deviations are caused by the same regression fault or change. Our study is about determining whether clustering could serve as an accurate strategy for grouping regression test deviations according to the faults or changes that caused them. We based our clustering strategy on the EM clustering algorithm because it does not require the number of clusters to be determined beforehand, but provides the opportunity to set an upper limit for it. Such an algorithm is fed with information regarding various aspects of the database elements being manipulated, and properties of the test cases exhibiting deviations, which is then used to compute distances between deviations.

We conducted a case study, in a real development setting, based on a large, critical database application developed by the Norwegian Tax Department. Four concrete regression test campaigns, covering three different parts of the system under test, were executed and analyzed by the test engineers in the project. That is, they executed the regression tests, and inspected all deviations from the test campaigns, while categorizing them into regression faults or changes. We then applied our proposed clustering strategy on different combinations of input data, and evaluated the accuracy of the deviation grouping for each type of input combination to determine what information was relevant for our objective. The results show that, for three out of the four test campaigns, clustering accuracy was perfect for at least one type of input. Additionally, four types of input yielded homogenous clusters (containing only one type of deviations) for all test campaigns. Homogenous clusters are our main priority as they imply that a tester inspecting one deviation from each cluster (one-per-cluster sampling) is certain to cover

all regression faults and changes causing deviations. Overall, the information that led to the most accurate deviation clustering was the combination of columns and operations in the deviation output, indicating differences in columns and operations being manipulated across versions for a given test case. That specific combination of input resulted in perfect accuracy for two of the test campaigns, while achieving perfectly homogeneous clusters for the two others, with a limited degree of dispersion across more than one cluster for common-cause deviations. In terms of inspection effort, for the four test campaigns under study, and when using a one-per-cluster sampling strategy, it was estimated that testers would only have to analyze a very small percentage of deviations, specifically three out of 48, two out of 47, two out of 84, and five out of 43, respectively, while still covering all regression faults and changes. This change is a dramatic increase in efficiency for deviation analysis in regression testing, and a way to achieve much higher confidence when analyzing deviations under time constraints. Such results are more specifically important for scaling regression testing to large database applications.

A possible future addition could be to help the tester understand the relation between model properties and the clusters. For instance, we could investigate whether it is possible to infer relations such as that all deviations in *Cluster A* relate to the same equivalence class for one or more model properties, and are the only ones in the test suites to do so. That would strongly indicate the types of properties that cause different types of deviations, which would further help the tester or developer when looking for the cause of regression test failure.

Acknowledgment

Lionel Briand was supported by the National Research Fund, Luxembourg (FNR/P10/03), while Erik Rogstad was supported by the ATOS project, a joint project of The Norwegian Tax Department and Simula Research Laboratory. The authors would also like to thank Cu Nguyen and Claudia Thür for their feedback on the paper.

Appendix - Entropy measurement details

This appendix contains the details of the clustering process per regression test, namely how the deviations are distributed across clusters, the deviation entropy per deviation type, and the cluster entropy per cluster.

- Table 3.9 shows entropy details for test campaign 1.
- Table 3.10 shows entropy details for test campaign 2.

- Table 3.11 shows entropy details for test campaign 3.
- Table 3.12 shows entropy details for test campaign 4.

Table 3.9: Entropy details for Test 1: For each type of input the deviation entropy is evaluated per type of deviation and the cluster entropy is evaluated per cluster

			Deviation Entropy				Cluster Entropy										Total Entropy		
		Test 1	Number of clusters	Fault 49166 Distribution	Entropy	Fault 49167 Distribution	Entropy	Cluster 0 Distribution	Entropy	Cluster 1 Distribution	Entropy	Cluster 2 Distribution	Entropy	Cluster 3 Distribution	Entropy	Cluster 4 Distribution	Entropy	Total Fault Entropy	Total Cluster Entropy	Total Entropy
Cluster input	Independent	Tables	5	(30,6,0,4,1)	-0,827	(0,0,7,0,0)	0,000	(30,0)	0,000	(6,0)	0,000	(0,7)	0,000	(4,0)	0,000	(1,0)	0,000	-0,827	0,000	-0,827
		Columns	3	(4,35,2)	-0,509	(6,0,1)	-0,410	(4,6)	-0,673	(35,0)	0,000	(2,1)	-0,637					-0,920	-1,310	-2,229
		Operations	3	(4,6,31)	-0,720	(0,7,0)	0,000	(4,0)	0,000	(6,7)	-0,690	(31,0)	0,000					-0,720	-0,690	-1,410
		Model Property Values	2	(9,32)	-0,526	(7,0)	0,000	(9,7)	-0,685	(32,0)	0,000							-0,526	-0,685	-1,212
	Pair	Tables and Columns	3	(35,6,0)	-0,416	(0,1,6)	-0,41	(35,0)	0,000	(6,1)	-0,41	(0,6)	0					-0,826	-0,410	-1,237
		Tables and Operations	3	(35,6,0)	-0,416	(0,0,7)	0	(35,0)	0,000	(6,0)	0	(0,7)	0					-0,416	0,000	-0,416
		Tables and Model Property Values	5	(6,9,9,0,17)	-1,312	(1,0,0,6,0)	-0,410	(6,1)	-0,410	(9,0)	0,000	(9,0)	0,000	(0,6)	0,000	(17,0)	0,000	-1,722	-0,410	-2,132
		Columns and Operations	3	(35,6,0)	-0,416	(0,0,7)	0,000	(35,0)	0,000	(6,0)	0,000	(0,7)	0,000					-0,416	0,000	-0,416
		Columns and Model Property Values	3	(6,13,22)	-0,979	(7,0,0)	0,000	(6,7)	-0,690	(13,0)	0,000	(22,0)	0,000					-0,979	-0,690	-1,670
		Operations and Model Property Values	4	(5,1,30,5)	-0,832	(1,0,0,6)	-0,410	(5,1)	-0,451	(1,0)	0,000	(30,0)	0,000	(5,6)	-0,689			-1,242	-1,140	-2,382
	Triple	Tables, Columns and Operations	5	(31,4,0,4,2)	-0,813	(0,0,7,0,0)	0,000	(31,0)	0,000	(4,0)	0,000	(0,7)	0,000	(4,0)	0,000	(2,0)	0,000	-0,813	0,000	-0,813
		Tables, Columns and Model Property Values	5	(2,5,10,0,24)	-1,062	(0,0,0,7,0)	0,000	(2,0)	0,000	(5,0)	0,000	(10,0)	0,000	(0,7)	0,000	(24,0)	0,000	-1,062	0,000	-1,062
		Tables, Operations and Model Property Values	3	(33,0,8)	-0,494	(0,7,0)	0	(33,0)	0,000	(0,7)	0	(8,0)	0					-0,494	0,000	-0,494
		Columns, Operations and Model Property Values	3	(35,0,6)	-0,416	(0,2,5)	-0,598	(35,0)	0,000	(0,2)	0,000	(6,5)	-0,689					-1,015	-0,689	-1,704
	All	Tables, Columns, Operations and Model Property Values	3	(6,35,0)	-0,416	(0,0,7)	0,000	(6,0)	0,000	(35,0)	0,000	(0,7)	0,000					-0,416	0,000	-0,416

Table 3.10: Entropy details for Test 2: For each type of input the deviation entropy is evaluated per type of deviation and the cluster entropy is evaluated per cluster

	Test 2	Number of clusters	Fault 49451		Change 49446		Cluster 0		Cluster 1		Cluster 2		Cluster 3		Cluster 4		Total Deviation Entropy	Total Cluster Entropy	Total Entropy
			Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy			
Cluster input: Independent	Tables	4	(0,16,0,3)	-0,436	(14,0,14,0)	-0,693	(0,14)	0,000	(16,0)	0,000	(0,14)	0,000	(3,0)	0,000			-1,129	0,000	-1,129
	Columns	3	(0,18,1)	-0,206	(24,4,0)	-0,410	(0,24)	0,000	(18,4)	-0,474	(1,0)	0,000					-0,616	-0,474	-1,090
	Operations	2	(0,19)	0,000	(24,4)	-0,410	(0,24)	0,000	(19,4)	-0,462							-0,410	-0,462	-0,872
	Model Property Values	3	(0,9,10)	-0,692	(14,0,14)	-0,693	(0,14)	0,000	(9,0)	0,000	(10,14)	-0,679					-1,385	-0,679	-2,064
Pair	Tables and Columns	3	(0,19,0)	0,000	(14,0,14)	-0,693	(0,14)	0	(19,0)	0	(0,14)	0,000					-0,693	0,000	-0,693
	Tables and Operations	5	(0,0,16,0,3)	-0,436	(10,4,0,14,0)	-0,992	(0,10)	0,000	(0,4)	0	(16,0)	0	(0,14)	0	(3,0)	0	-1,428	0,000	-1,428
	Tables and Model Property Values	5	(4,2,6,0,7)	-1,297	(6,6,0,16,0)	-0,980	(4,6)	-0,673	(2,6)	-0,562	(6,0)	0,000	(0,16)	0,000	(7,0)	0,000	-2,277	-1,235	-3,512
	Columns and Operations	2	(19,0)	0,000	(0,28)	0,000	(19,0)	0,000	(0,28)	0,000							0,000	0,000	0,000
	Columns and Model Property Values	3	(19,0,0)	0,000	(0,16,12)	-0,683	(19,0)	0,000	(0,16)	0,000	(0,12)	0,000					-0,683	0,000	-0,683
	Operations and Model Property Values	4	(6,6,0,7)	-1,096	(12,0,16,0)	-0,683	(6,12)	-0,637	(6,0)	0,000	(0,16)	0,000	(7,0)	0,000			-1,779	-0,637	-2,415
Triple	Tables, Columns and Operations	5	(0,0,16,0,3)	-0,436	(10,4,0,14,0)	-0,992	(0,10)	0,000	(0,4)	0,000	(16,0)	0,000	(0,14)	0,000	(3,0)	0,000	-1,428	0,000	-1,428
	Tables, Columns and Model Property Values	5	(0,0,10,0,9)	-0,692	(10,4,0,14,0)	-0,992	(0,10)	0,000	(0,4)	0,000	(10,0)	0,000	(0,14)	0,000	(9,0)	0,000	-1,684	0,000	-1,684
	Tables, Operations and Model Property Values	3	(19,0,0)	0,000	(0,20,8)	-0,598	(19,0)	0,000	(0,20)	0,000	(0,8)	0,000					-0,598	0,000	-0,598
	Columns, Operations and Model Property Values	5	(0,0,10,0,9)	-0,692	(10,4,0,14,0)	-0,992	(0,10)	0,000	(0,4)	0,000	(10,0)	0,000	(0,14)	0,000	(9,0)	0,000	-1,684	0,000	-1,684
All	Tables, Columns, Operations and Model Property Values	5	(0,0,16,0,3)	-0,436	(11,5,0,8,4)	-1,311	(0,11)	0,000	(0,5)	0,000	(16,0)	0,000	(0,8)	0	(3,4)	-0,683	-1,747	-0,683	-2,430

Table 3.11: Entropy details for Test 3: For each type of input the deviation entropy is evaluated per type of deviation and the cluster entropy is evaluated per cluster

	Test 3	Number of clusters	Change 45024		Fault 50900		Cluster 0		Cluster 1		Cluster 2		Cluster 3		Cluster 4		Cluster 5		Total Deviation Entropy	Total Cluster Entropy	Total Entropy
			Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy			
Cluster input: Independent	Tables	2	(56,0)	0,000	(0,28)	0,000	(56,0)	0,000	(0,28)	0,000									0,000	0,000	0,000
	Columns	5	(50,0,2,2,2)	-0,458	(0,28,0,0,0)	0,000	(50,0)	0,000	(0,28)	0,000	(2,0)	0,000	(2,0)	0,000	(2,0)	0,000			-0,458	0,000	-0,458
	Operations	2	(56,0)	0,000	(0,28)	0,000	(56,0)	0,000	(0,28)	0,000									0,000	0,000	0,000
	Model Property Values	1	(56)	0,000	(28)	0,000	(56,28)	-0,637											0,000	-0,637	-0,637
Cluster input: Pair	Tables and Columns	5	(50,0,2,2,2)	-0,458	(0,28,0,0,0)	0,000	(50,0)	0,000	(0,28)	0,000	(2,0)	0,000	(2,0)	0,000	(2,0)	0,000			-0,458	0,000	-0,458
	Tables and Operations	2	(56,0)	0,000	(0,28)	0,000	(56,0)	0,000	(0,28)	0									0,000	0,000	0,000
	Tables and Model Property Values	2	(56,0)	0,000	(0,28)	0,000	(56,0)	0,000	(0,28)	0,000									0,000	0,000	0,000
	Columns and Operations	2	(56,0)	0,000	(0,28)	0,000	(56,0)	0,000	(0,28)	0,000									0,000	0,000	0,000
	Columns and Model Property Values	4	(14,28,14,0)	-1,040	(0,0,0,28)	0,000	(14,0)	0,000	(28,0)	0,000	(14,0)	0,000	(0,28)	0,000					-1,040	0,000	-1,040
	Operations and Model Property Values	3	(28,0,28)	-0,693	(0,28,0)	0,000	(28,0)	0,000	(0,28)	0,000	(28,0)	0,000							-0,693	0,000	-0,693
	Tables, Columns and Operations	2	(0,56)	0,000	(28,0)	0,000	(0,28)	0,000	(56,0)	0,000									0,000	0,000	0,000
	Tables, Columns and Model Property Values	2	(56,0)	0,000	(0,28)	0,000	(56,0)	0,000	(0,28)	0,000									0,000	0,000	0,000
Triple	Tables, Operations and Model Property Values	6	(1,16,3,23,0,13)	-1,291	(12,0,0,0,16,0)	-0,683	(1,12)	-0,271	(16,0)	0	(3,0)	0	(23,0)	0	(0,16)	0	(13,0)	0	-1,974	-0,271	-2,245
	Columns, Operations and Model Property Values	6	(4,20,0,0,24,8)	-1,197	(0,0,14,14,0,0)	-0,693	(4,0)	0,000	(20,0)	0,000	(0,14)	0,000	(0,14)	0,000	(24,0)	0,000	(8,0)	0,000	-1,890	0,000	-1,890
All	Tables, Columns, Operations and Model Property Values	3	(28,0,28)	-0,693	(0,28,0)	0,000	(28,0)	0,000	(0,28)	0,000	(28,0)	0,000							-0,693	0,000	-0,693

Table 3.12: Entropy details for Test 4: For each type of input the deviation entropy is evaluated per type of deviation and the cluster entropy is evaluated per cluster

	Test 4	Number of clusters	Fault 50030		Change 50031		Change 48580		Cluster 0		Cluster 1		Cluster 2		Cluster 3		Cluster 4		Total Deviation Entropy	Total Cluster Entropy	Total Entropy
			Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy	Distribution	Entropy			
Cluster input: Independent	Tables	3	(0,0,3)	0,000	(0,10,0)	0,000	(30,0,0)	0,000	(0,0,30)	0,000	(0,10,0)	0,000	(3,0,0)	0,000					0,000	0,000	0,000
	Columns	5	(0,0,3,0,0)	0,000	(0,0,10,0,0)	0,000	(5,9,0,12,4)	-1,295	(0,0,5)	0,000	(0,0,9)	0,000	(3,10,0)	-0,540	(0,0,12)	0,000	(0,0,4)	0,000	-1,295	-0,540	-1,835
	Operations	5	(0,0,0,0,3)	0,000	(0,10,0,0,0)	0,000	(25,0,3,2,0)	-0,563	(0,0,25)	0,000	(0,10,0)	0,000	(0,0,3)	0,000	(0,0,2)	0,000	(3,0,0)	0,000	-0,563	0,000	-0,563
	Model Property Values	3	(1,0,2)	-0,637	(1,0,9)	-0,325	(19,11,0)	-0,657	(1,1,19)	-0,381	(0,0,11)	0,000	(2,9,0)	-0,474					-1,619	-0,855	-2,473
Pair	Tables and Columns	5	(0,0,0,0,3)	0,000	(0,0,10,0,0)	0,000	(12,9,0,9,0)	-1,089	(0,0,12)	0,000	(0,0,9)	0	(0,10,0)	0,000	(0,0,9)	0	(3,0,0)	0	-1,089	0,000	-1,089
	Tables and Operations	5	(0,0,0,0,3)	0,000	(0,10,0,0,0)	0,000	(12,0,3,15,0)	-0,943	(0,0,12)	0,000	(0,10,0)	0	(0,0,3)	0	(0,0,15)	0	(3,0,0)	0	-0,943	0,000	-0,943
	Tables and Model Property Values	2	(3,0)	0,000	(10,0)	0,000	(0,30)	0,000	(3,10,0)	-0,540	(0,0,30)	0,000							0,000	-0,540	-0,540
	Columns and Operations	5	(0,0,0,0,3)	0,000	(0,0,10,0,0)	0,000	(12,9,0,9,0)	-1,089	(0,0,12)	0,000	(0,0,9)	0,000	(0,10,0)	0,000	(0,0,9)	0,000	(3,0,0)	0,000	-1,089	0,000	-1,089
	Columns and Model Property Values	4	(0,0,3,0)	0,000	(0,0,10,0,0)	0,000	(12,11,0,7)	-1,074	(0,0,12)	0,000	(0,0,11)	0,000	(3,10,0)	-0,540	(0,0,7)	0,000			-1,074	-0,540	-1,614
	Operations and Model Property Values	2	(0,3)	0,000	(0,10)	0,000	(30,0)	0,000	(0,0,30)	0,000	(3,10,0)	-0,540							0,000	-0,540	-0,540
Triple	Tables, Columns and Operations	5	(0,0,0,0,3)	0,000	(0,0,10,0,0)	0,000	(12,9,0,9,0)	-1,089	(0,0,12)	0,000	(0,0,9)	0,000	(0,10,0)	0,000	(0,0,9)	0,000	(3,0,0)	0,000	-1,089	0,000	-1,089
	Tables, Columns and Model Property Values	3	(3,0,0)	0,000	(10,0,0)	0,000	(0,18,12)	-0,673	(3,10,0)	-0,540	(0,0,18)	0,000	(0,0,12)	0,000					-0,673	-0,540	-1,213
	Tables, Operations and Model Property Values	4	(0,0,3,0)	0,000	(0,0,10,0)	0,000	(12,4,0,14)	-0,991	(0,0,12)	0,000	(0,0,4)	0	(3,10,0)	-0,54	(0,0,14)	0			-0,991	-0,540	-1,531
	Columns, Operations and Model Property Values	2	(3,0)	0,000	(10,0)	0,000	(0,30)	0,000	(3,10,0)	-0,540	(0,0,30)	0,000							0,000	-0,540	-0,540
All	Tables, Columns, Operations and Model Property Values	4	(0,0,0,3)	0,000	(0,0,0,10)	0,000	(9,12,9,0)	-1,089	(0,0,9)	0,000	(0,0,12)	0,000	(0,0,9)	0,000	(3,10,0)	-0,540			-1,089	-0,540	-1,629

7 References

[1] Md. Junaid Arafeen and Hyunsook Do. "Test Case Prioritization Using Requirements-Based Clustering." In: *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. 2013, pp. 312–321.

[2] Andrea Arcuri and Lionel Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering." In: *Software Engineering (ICSE), 2011 33rd International Conference on*. 2011, pp. 1–10.

[3] Markus Borg and Per Runeson. "IR in Software Traceability: From a Bird's Eye View." In: *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*. 2013, pp. 243–246.

[4] T. Calinski and J. Harabasz. "A dendrite method for cluster analysis." In: *Communications in Statistics* 3.1 (1974), pp. 1–27.

[5] Ryan Carlson, Hyunsook Do, and Anne Denton. "A clustering approach to improving test case prioritization: An industrial case study." In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. 2011, pp. 382–391.

[6] Songyu Chen et al. "Using semi-supervised clustering to improve regression test selection techniques." In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. 2011, pp. 1–10.

[7] William Dickinson, David Leon, and Andy Podgurski. "Finding failures by cluster analysis of execution profiles." In: *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*. 2001, pp. 339–348.

[8] Nicholas DiGiuseppe and James A. Jones. "Concept-based failure clustering." In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 2012, 29:1–29:4.

[9] Colleen Graham et al. *Market Share: All Software Markets, Worldwide, 2013*. 2014. URL: `https://www.gartner.com/doc/2695617` (visited on 06/15/2014).

[10] Robert M. Gray. *Entropy and Information Theory*. 2nd. New York: Springer, 2011.

[11] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A methodology for controlling the size of a test suite." In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (July 1993), pp. 270–285.

[12] Mary Jean Harrold and Alessandro Orso. "Retesting software during development and maintenance." In: *Frontiers of Software Maintenance, 2008. FoSM 2008.* 2008, pp. 99–108.

[13] Chien-Hsin Hsueh, Yung-Pin Cheng, and Wei-Cheng Pan. “Intrusive Test Automation with Failed Test Case Clustering.” In: *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*. 2011, pp. 89–96.

[14] Leonard Kaufman and Peter J. Rousseeuw. *Finding groups in data - An introduction to cluster analysis*. United States: John Wiley and Sons Inc., 2005.

[15] Yue Liu et al. “User-Session-Based Test Cases Optimization Method Based on Agglutinate Hierarchy Clustering.” In: *Proceedings of the 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*. ITHINGSCPSCOM ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 413–418.

[16] Geoffrey J. McLachlan and Thriyambakam Krishnan. *The EM algorithm and extensions*. 2nd. Hoboken, NJ: Wiley, 2008. XXVII, 359.

[17] Saul B. Needleman and Christian D. Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins.” In: *Journal of Molecular Biology* 48.3 (1970), pp. 443 –453.

[18] Saeed Parsa, Alireza Khalilian, and Yalda Fazlalizadeh. “A new algorithm to Test Suite Reduction based on cluster analysis.” In: *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*. 2009, pp. 189–193.

[19] Kai Petersen and Claes Wohlin. “Context in industrial software engineering research.” In: *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. 2009, pp. 401–404.

[20] Andy Podgurski and Charles Yang. “Partition testing, stratified sampling, and cluster analysis.” In: *SIGSOFT Softw. Eng. Notes* 18.5 (Dec. 1993), pp. 169–181.

[21] Andy Podgurski et al. “Automated support for classifying software failure reports.” In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. 2003, pp. 465–475.

[22] Andy Podgurski et al. “Estimation of software reliability by stratified sampling.” In: *ACM Trans. Softw. Eng. Methodol.* 8.3 (July 1999), pp. 263–283.

[23] Erik Rogstad, Lionel Briand, and Richard Torkar. “Test Case Selection for Black-Box Regression Testing of Database Applications.” In: *Information and Software Technology (IST)* 31.6 (2013), pp. 676–686.

[24] Erik Rogstad et al. “Industrial Experiences with Automated Regression Testing of a Legacy Database Application.” In: *27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 362–371.

[25] Gregg Rothermel and Mary Jean Harrold. “A safe, efficient regression test selection technique.” In: *ACM Trans. Softw. Eng. Methodol.* 6.2 (Apr. 1997), pp. 173–210.

[26] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering.” In: *Empirical Softw. Engg.* 14.2 (2009), pp. 131–164.

[27] P. G. Sapna and Hrushikesha Mohanty. “Clustering test cases to achieve effective test selection.” In: *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*. A2CWiC ’10. Coimbatore, India: ACM, 2010, 15:1–15:8.

[28] Cristian Simons and Emerson Cabrera Paraiso. “Regression test cases prioritization using Failure Pursuit Sampling.” In: *Intelligent Systems Design and Applications (ISDA), 2010 10th International Conference on*. 2010, pp. 923–928.

[29] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. “Test case comparison and clustering using program profiles and static execution.” In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC/FSE ’09. Amsterdam, The Netherlands: ACM, 2009, pp. 293–294.

[30] András Vargha and Harold D. Delaney. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong.” In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[31] Yabin Wang et al. “Using Weighted Attributes to Improve Cluster Test Selection.” In: *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. 2012, pp. 138–146.

[32] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd. San Francisco: Morgan Kaufmann, 2005.

[33] Shali Yan et al. “A Dynamic Test Cluster Sampling Strategy by Leveraging Execution Spectra Information.” In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. 2010, pp. 147–154.

[34] Robert K. Yin. *Case study research: Design and methods*. 3rd. London: Sage, 2003.

[35] Shin Yoo and Mark Harman. “Regression Testing Minimisation, Selection and Prioritisation: A Survey.” In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

[36] Shin Yoo et al. “Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge.” In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ISSTA ’09. Chicago, IL, USA: ACM, 2009, pp. 201–212.

[37] Chen Zhang et al. “An Improved Regression Test Selection Technique by Clustering Execution Profiles.” In: *Quality Software (QSIC), 2010 10th International Conference on*. 2010, pp. 171–179.