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

Erik Rogstad[a], Lionel Briand[b]

[a]*Simula Research Laboratory, Oslo*
[b]*SnT Centre, University of Luxembourg, Luxembourg*

Abstract

Testing and, more specifically, the regression testing of database applications is highly challenging and costly. One can rely on production data or generate synthetic data, for example based on combinatorial techniques or operational profiles. Both approaches have drawbacks and advantages. Automating testing with production data is impractical and combinatorial test suites might not be representative of system operations.

In this paper, based on a large scale case study in a representative development environment, we explore the cost and effectiveness of various approaches and their combination for the regression testing of database applications, based on production data and synthetic data generated through classification tree models of the input domain.

The results confirm that combinatorial test suite specifications bear little relation to test suite specifications derived from the system operational profile. Nevertheless, combinatorial testing strategies are effective, both in terms of the number of regression faults discovered but also, more surprisingly, in terms of the importance of these faults. However, our study also shows that relying solely on synthesized test data derived from test models could lead to important faults slipping to production. Thus, we recommend that testing on production data and combinatorial testing be combined to achieve optimal results.

Keywords: Regression testing, Database applications, Classification tree modeling, Combinatorial testing, Operational profile testing, Test data generation

1. Introduction

In large database applications, constructing synthetic test data is often deemed too time-consuming because of the large number of data dependencies in a typical relational database. Thus, a common strategy when testing database applications is to rely on production data for testing, i.e. data from the system operation. However, available production data may not satisfy all the requirements of the test plan (test specifications), thus forcing the testers to manipulate and/or extend the production data to make it adequate. Both the task of identifying appropriate test data and subsequently manipulating it if it does not exactly fit the needs, is a tedious process for the testers. The use of production data nevertheless remains common practice due to the lack of alternatives.

In order to achieve effective test automation, for example to support regression testing, we would like to rely on test suites conforming to clear test strategies, i.e. devised in a structured and systematic manner, to ensure predictability of test results both in terms of coverage and fault revealing power. Thus, automatically generating synthetic test data becomes necessary. Yet again, when generating test data, a wide range of possibilities opens up. Rather than finding appropriate production data, the challenge shifts to limiting the amount of test data to generate to a proper sized test suite matching the available test budget, while still ensuring predictable quality. Combinatorial testing is an attractive strategy for generating compact n-way test suite specifications [1, 2], and consequently a natural starting point for test data generation. Classification tree modeling is a common approach to combinatorial test design [3], in which the input domain of the system under test is modeled as a classification tree, which in turn is used to generate a combinatorial test suite specification [4].

However, based on our experience, combinatorial test suite specifications do not necessarily align well with system usage and the benefits in terms of risk reduction resulting from testing are therefore unclear. In other

Email addresses: erikrog@simula.no (), (Lionel Briand)

words, when your test model is fairly complex, a compact combinatorial test suite may mostly consist of test cases that are rarely or never executed in the real operation of the system. To ensure that tests match meaningful and representative executions of the system, we can analyze the operational profile of some of the functional areas of the system under test, on which basis we can generate test suite specifications directly aligned with system usage. Alternatively, we can use the operational profile to weigh the properties in our combinatorial models (classification trees), which is then used to generate more representative combinatorial test suites.

In this paper we present an empirical investigation, in a real and representative database application development environment, of various strategies for the selection and generation of test data for database applications. We assess production data and generate synthetic test data following various combinatorial strategies based on classification tree models. We focus on system level regression testing and run actual regression tests to compare the fault detection rate of each strategy. Our case study focuses on the following questions:

- What combinatorial test coverage can be expected from production data? It is important to understand the scope of production data if they are to be used for testing, and also help determine whether additional data need to be synthesized for achieving satisfactory testing.
- How effective are synthesized combinatorial regression test suites, based on pair-wise and three-wise coverage, at assessing and eliminating the risks of failure? Combinatorial test suites tend to be compact, as they target model coverage with a small set of test cases. It is then interesting to explore to what extent the compact test suites are representative of the usage of the system being tested. This will determine whether they are able to detect relevant faults, that is faults that are likely to manifest themselves in practice and present significant risks.
- Can operational profile analysis help guide the synthesis of more effective regression tests? One important question is whether we can use an operational profile analysis to weigh model property values, in order to generate more representative combinatorial test suites. As an alternative to combinatorial testing, we must also investigate whether synthesized data based on an operational profile is an effective regression test strategy, or a complement to combinatorial testing.
- How do test strategies compare, in terms of their ability to detect faults, during regression testing and when generated according to: (1) Synthesized combinatorial test data (pair-wise and three-wise), (2) The operational profile of the system, (3) Synthesized pair-wise test data weighed based on the operational profile, and (4) production data. Whether to test on production data or synthesized data often comes down to what is more practical in a given context. However, assessing both alternatives and comparing them is interesting to understand the consequences of choosing one over the other or combining them. Furthermore, it is interesting to investigate whether synthesized data based on the operational profile can detect similar faults as tests derived from production data.

The remainder of the paper is organized as follows: Section 2 provides information about the industrial context of our work, along with the background and motivation for our case study. Section 3 outlines a practical approach to generate test data, based on classification tree models, and the matching of production data against these models. The design of the case study is presented in Section 4, along with empirical results, discussions, and threats to validity. Related work is reported in Section 5, before drawing conclusions in Section 6.

2. Background and motivation

The Norwegian Tax Department maintains several large database systems. For example, the Norwegian tax accounting system, SOFIE, serves more than 3,000 end users (taxation officers) and handle yearly tax revenues of approximately 600 Billion NOK. Common to these systems is that they process large amounts of data, and accordingly the business logic of the systems is organized into batch programs to ensure efficient data processing. Batch programs typically process large sets of input data and run to completion without human intervention. This makes them suitable for automated transaction handling, and the batches are thereby often scheduled to run periodically. As being a part of the Tax Department portfolio, the systems are subject to changes in taxation laws, along with usual maintenance activities, thus causing continuous changes to the batch programs. It is vital for the tax department to avoid releasing faults upon changes and maintain system quality to preserve taxpayers' confidence. Nevertheless, regression faults in the batch programs have been a struggle throughout the years. Further, the batch programs process large amounts of data, spanning a wide variety of

possible test cases, which makes manual testing inadequate to support quality assurance in the context of frequent changes.

2.1. Automated regression testing

To address some of the issues mentioned above, we proposed a partly automated regression test procedure and tool (DART) tailored to database applications [5]. 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 regression faults.

The strength of this approach is that it provides the ability to verify the entire set of test data executed by a batch automatically. As an example, lets 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.

2.2. Observations regarding test data

Throughout initial phases of applying the proposed methodology for automated regression testing, we made a number of observations regarding test data, that formed the basis for the work presented in this paper. Our case study is further elaborated in Section 4, whereas the remainder of this section will elaborate on the observations motivating it.

2.2.1. Testing using production data

In a system like SOFIE, like in many other database applications, there are vast amounts of data available from the production environment of the system. Consequently, the testing of SOFIE has traditionally been heavily relying on the use of anonymized production data. 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 when running tests.

This was our starting point for the regression testing of the batch programs. However, initial regression tests on production data indicated significant redundancy in the data and very unpredictable test coverage. Redundancy was visible when conducting the deviation analysis, as many test cases executed the system in a similar way and triggered the same fault, thus causing many redundant deviations to inspect. The unpredictable test coverage became evident when we ran the same regression test (same batch program versions) using three different sets of production data and found some overlapping faults, but also many distinct ones in each set. These observations led us to further investigate the applicability of production data for regression testing. Given the arbitrariness of relying on production data for testing (the availability and characteristics of data vary over time), what level of redundancy and coverage should be expected? Thus, research question one (RQ1) is about assessing the common practice of using production data for testing, as further specified in section 4.1. We believe this question is important, as it is common practice in many governmental institutions and private enterprises maintaining large database applications, to rely on production data when testing. To provide industrial figures regarding the redundancy and coverage from such data is important to better understand the strength and weaknesses of such an approach. Such increased awareness could also help influence decisions on whether to take on a more ambitious approach for generating test data, and investigate the benefits and challenges of doing so.

2.2.2. Generating synthetic test data

The observations regarding the high level of redundancy and unpredictable coverage in production data, along with other impracticalities, motivated us to start generating synthetic test data. As will be elaborated further in Section 3, we rely on classification tree models to model the input domain of the system under test. Classification tree modeling is a category-partition-based modeling technique, where properties of the input domain is modeled in a hierarchical tree structure [6, 3]. Each property is further split into equivalence classes, each one representing a distinct range of values in the property domain. We use these models to drive the test data generation, and when we started testing with synthetic test data we used algorithms to generate pairwise combinatorial test suites, to ensure good model

coverage (all pairs of equivalence classes are covered), while minimizing the test suites. We then ran regression tests, found faults and reported them, but found that most of them were rejected by software engineers (not corrected) as they were deemed too rare to happen often enough in the operation of the system to justify the cost of fixing them. In other words, we found theoretically viable functional faults, that bore little relevance to the practical operation of the system. Whereas this experience in isolation does not rule out pair-wise testing as a plausible test technique, it certainly made us think twice about whether pair-wise test suites were adequate to support regression testing. However, there is no harm in finding faults that are regarded as unimportant, as long as it does not come at the cost of finding the more important ones.

2.2.3. Analyzing the operational profile of the system under test

Given the significant number of unimportant faults found with pair-wise testing, we realized that it would make sense to complement pair-wise testing, with test suites that better reflected the operations of the system. Thus, we conducted an analysis of the operational profile of a part of the system under test, to better understand which executions that are most common in the operation of the system [7]. We did this by matching actual tax calculations for more than four million taxpayers with the classification tree model, in order to learn the number of test cases matching the different model partitions (abstract test cases) and also the frequency of each equivalence class of a property in the operation of the system. Thus, in our context an operational profile is represented as a probability distribution of equivalence classes and model partitions for a given classification tree model.

We found that the five most frequent model partitions contained approximately 1,400,000, 1,200,000, 450,000, 260,000 and 230,000 tax calculations, respectively. Furthermore, the ten most frequent model partitions in the operational profile of the system represented 93% of all the tax calculations, while the fifty most frequent ones represented 98% of all the tax calculations. This showed that by including relatively few test cases, we would ensure test coverage of the most frequent system executions, which would hopefully prevent faults from affecting the vast majority of taxpayers. Based on the observations made regarding pair-wise test suites and the analysis of the operational profile, we wanted to investigate how the partitions (test cases) generated by the pair-wise and three-wise algorithms aligned with the partitions from a test suite based on the operational profile of the system, and also their relative fault detection revealing power. This led to RQ2 and RQ3, which are about evaluating the practice of pair-wise and three-wise testing in terms of how realistic their test cases are, and then compare the ability of each test strategy (combinatorial test suites and test suites based on the operational profile, both synthetically generated and tests based on production data) to find regression faults, and the importance of the faults reported.

3. Proposed method to systematically control test data using classification tree models

Our approach to regression testing is black-box, relying on widely used classification tree models to model the input domain of the system under test. We chose to focus on black-box (specification-based) testing, using classification tree models, in order to obtain a practical and scalable solution for regression testing. In many situations, white-box testing is not practical due to lack of proper tool support, or not even applicable if there is no direct access to the source code or third party components. Then a black-box approach, based on the system specifications, must be adopted [5, 8, 9].

We use these test models to (1) systematically drive the synthetic test data generation and (2) to match production data against the model in order to determine coverage, either a) to select production data for testing or (b) to analyze the operational profile of a *functional domain* of the system under test, i.e. a particular functional area of the system under test. In the remainder of this section we will elaborate our practical approach for doing so, after first introducing the basic concepts of classification tree modeling.

3.1. Classification tree models

As mentioned in Section 2.2.2, classification tree modeling is a category-partition-based modeling technique [6, 3]. Classification tree modeling is typically used for modeling configuration parameters [10, 11, 12], or the input domain of the system under test, i.e. input parameters [13] or properties of the system under test [14]. In our case we model properties of the input domain in a hierarchical tree structure and split each property domain or range into equivalence classes, which represent sub-domains of values tailored for test purposes. The objective is to capture the variability in the input domain of the system under test, and then ensure we have tests covering a wide spectrum of combinations of input values.

An artificial example of a classification tree model, made using the tool CTE XL [4], is given in Figure 1.

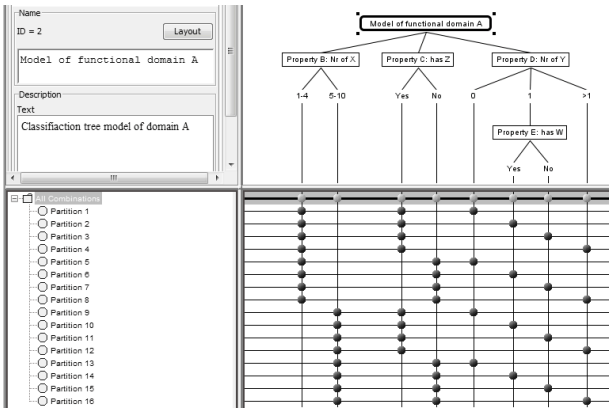

Figure 1: An example of a classification tree model in CTE XL and the generated partitions (combinations of equivalence classes) that form abstract test cases.

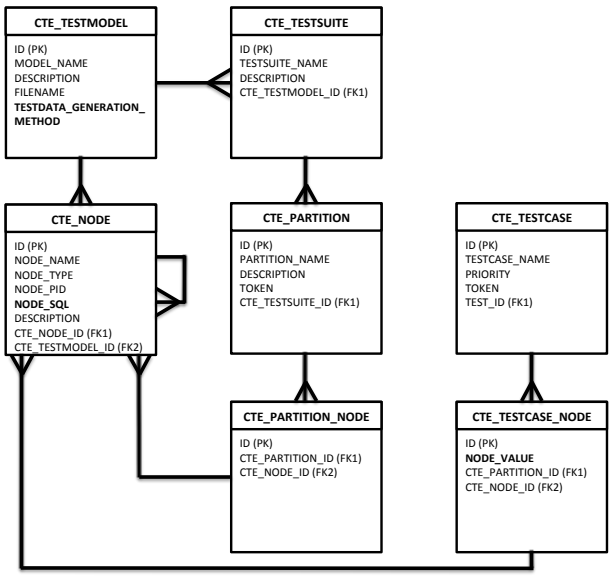

Figure 2: ER-diagram for CTE XL models in DART.

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. Equivalence classes could also be created for invalid ranges (e.g. < 0) [15]. However, as our regression test approach is targeted at testing system functionality, not robustness testing, we have not focused on invalid ranges in this paper.

The lower part of Figure 1 shows the model partitions. A partition is a specific combination of equivalence classes. Given the example model, a partition would be represented as the combination of three equivalence classes, namely one from Property B, one from Property C and one from Property D/E. For example a partition could be composed of the tuple *5-10*, *Yes*, *¿1*, which constitutes partition 12 in the model. Once the model is established, partitions can be generated. For example, partitions satisfying the pair-wise model coverage criteria can be generated, which ensures that every pair of equivalence classes are covered by at least one partition. For the simple model presented here, we have generated all possible combinations, and all resulting partitions are gathered in a set labeled "All Combinations". 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.

To be able to use the models for anything practical, we need to integrate them with our regression test tool, DART [5]. DART has its own database, where everything related to the regression tests are stored, i.e. test configurations, test executions and test results. We have extended the database schema to also 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 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 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.

3.2. Selecting test data based on test models

When using production data as basis for testing, the selected test data will vary between (regression) test campaigns. In order to remain systematic when testing, we use the test models (classification trees) 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 parti-

tions 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. We refer the reader to Rogstad et al. [8] for further details on that subject.

The classification tree models tend to be high-level functional representations of the input domain of the system under test. Indeed, the input space could be characterized in a more complete or refined way with more tree properties and more equivalence classes associated with these properties, to better capture the variability in functionality and behavior of the system. 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 filled 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 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 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. An alternative strategy would be to attach an SQL query to all leaf nodes in the model. It would be equally general and simpler to implement, as you would not have to traverse the entire tree structure, but rather execute all leaf node queries for all test cases. However, it would result in the execution of many, nearly identical queries, as the queries for each specific equivalence class underlying a model property is highly likely to be very similar. It would also, in general, result in more SQL mapping queries to generate for the tester. Therefore, we chose to base the mapping through SQL queries on the level of model properties (classifications). When the matching is done, 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.

3.3. Generating synthetic test data

The 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 synthetic test data. Using the same classification tree models, both for analyzing coverage of production data used for testing and for the generation of synthetic 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 synthetic 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 test data. Even a "simple" test case will most likely require a large extent of data populated in large parts of the relational 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. 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. In other words, the test data generator can generate any test case within the scope of a given classification tree model. It takes a test suite specification as input and generates test data satisfying the specification for each test case in the suite.

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 as time goes by.

To extend the generation of test data to new application domains, the mapping code between the classification tree model that interprets the model and calls on the test data API to populate test data, have to be implemented. Extensions to the test data API is also necessary if the new models require data in tables that are not yet implemented in the API. Additionally, configuration variables are needed, they need to be defined at the test suite specification and partition level and accounted for in the adapter code.

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. As an example, consider the classification tree model in Figure 1. The model has no knowledge about the system implementation, but rather capture variations in the input domain at a functional level. So the task of the mapping code is to map the model properties to actual system data in the database. In general in our context, the mapper needs to create a taxpayer, and then interpret each model property and call on the right API code to populate data accordingly. For example, *Property B: Nr of X* may be the number of children of the taxpayer, taking on two value ranges, 1-4 and 5-10. The generator will pick a random number in

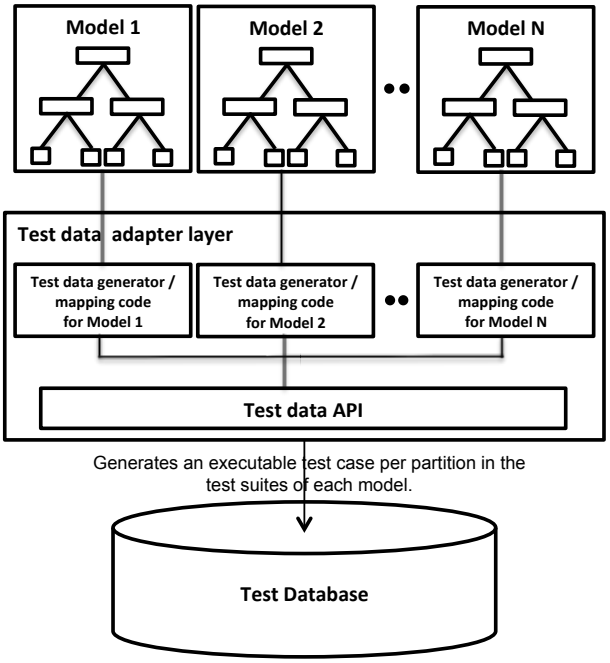

Figure 3: Synthetic test data is generated using a test data adapter layer, that interprets a model and uses a test data API to creates executable test cases in the test database.

the given range, but according to a given seed to ensure deterministic random values for data sets that should be comparable across regression tests by reusing the same seed. It will then create the specified number of persons in the database and link them as children of the taxpayer. Since no further specification was given on the children, i.e. no hierarchical sub-properties, their values are randomly generated, yet deterministic when needed. Furthermore, *Property C: has Z*, could be whether or not the taxpayer is entitled to a tax deduction of a certain percent for some reason. Depending on the value, i.e. *Yes* or *No* from the model, the properties are set accordingly in the database for the synthesized taxpayer.

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 except for basis data. 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.

In the case study presented in the next Section, we will explore several different strategies for generating test suite specifications from the model, and in turn executable regression test suites. Though *pair-wise*

and *three-wise* are common combinatorial testing techniques, we provide a short description of all strategies in our context.

Pair-wise (2W). The *pair-wise* generation criteria is satisfied if every possible pair of equivalence classes is covered by at least one test case specification in the resulting test suite specification. Following the example from Figure 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 [16, 17, 2].

Three-wise (3W). The *three-wise* generation criteria is satisfied if every possible triple of equivalence classes is covered by at least one test case specification in the resulting test suite specification. Following the example from Figure 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 [18, 19].

Weighed pair-wise (W2W). As for *pair-wise* the *weighed pair-wise* generation criteria is satisfied if every pair of equivalence classes is covered by at least one test case specification in the resulting test suite specification. However, pair-wise, whose only objective is to ensure pair-wise coverage, treats all equivalence classes equally (while accounting for constraints if present) and targets a minimal test suite ensuring adequate coverage. The weighed pair-wise approach, in contrast, accounts for the weights of the equivalence classes (how weights are assigned as explained below) while seeking pair-wise coverage. It does so by using the most likely combination of equivalence classes (partition) as the base test case (i.e. the starting point for pair-wise generation) and whenever alternative equivalence classes can be selected to equally contribute to the objective of pair-wise coverage, the algorithm will select the highest probability one [20].

Operational profile (OP). The operational profile test suite specification is generated by randomly selecting test case specifications according to the probability distribution across the equivalence classes. Its objective is to generate a test suite that aligns with system usage, rather than targeting a specific model coverage (e.g. pair-wise). The number of test case specifications to generate is specified by the user.

The weights to the equivalence classes used by the two latter generation strategies have been set on the basis of the operational profile analysis. We have matched actual tax executions with the classification tree models in the way we explained in Section 3.2 in order to learn the probability distribution of the equivalence classes in the model. Following the example model given in Figure 1, if 75 % of the data in the operation profile have the value *Yes* for *Property C: has Z*, then weights are set accordingly to each leaf node. These equivalence class weights are used to drive the generation of weighed pair-wise and operational profile test suite specifications [7].

4. Case study

This section outlines the case study design and execution according to the guidelines given by Runeson and Höst [21] and presents the results of the study. These guidelines are tailored for reporting case studies within Software Engineering, and thus suitable for our context.

4.1. Objective and research questions

This case study focuses on experiences and observations regarding regression testing of a large, representative industrial database application. The primary objective is to assess the practice of using production data for regression testing purposes and evaluate the alternative practice of using synthetic test data based on combinatorial test selection, weighed or not based on the system operational profile. Given the characteristics of the system and environment under study (as will be further elaborated in Section 4.2), we believe that our lessons learned can apply to many other contexts.

More specifically, the following research questions are addressed:

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 test engineers expect from production data used for testing, when matched against a realistic classification tree model, i.e. model developed by domain experts?
2. For a given classification tree model, how much redundancy can test engineers expect with production data, 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. to which extent do combinatorial test suite specifications represent common usage scenarios?

RQ3 Ultimately, we want to favor a test strategy that reveals important faults. By important, we mean faults with 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?

4.2. Case study context

Petersen and Wohlin [22] suggested a checklist for reporting context information in industrial software engineering research. This is important in order to provide the reader with contextual information for the results, and thus the possibility to consider whether to generalize them to another context. The checklist is complementary to the general guidelines given by Runeson and Höst [21] for reporting software engineering case studies. We will comply with this checklist when describing *the case* and the context influencing it. The case study concerns the regression testing of database applications, and thus important context elements are the system being tested, the people involved in the test process, and the practices, tools and techniques supporting the test effort. We describe these different elements (facets), as suggested by Petersen and Wohlin.

4.2.1. The subject system

The system under test in our case study is the Norwegian tax accounting system, SOFIE. As mentioned in Section 2, SOFIE serves more than 3,000 end users (taxation officers) and handles yearly tax revenues of approximately 600 Billion NOK. The system was developed as a customized application for the Norwegian tax department during the late 2000s and entered its maintenance phase in 2011.

The system holds large amounts of data as it serves the tax calculations for the entire nation of Norway, i.e. ten years of historical tax data for millions of taxpayers. Thus, performance related aspects have been the main quality driver during development, in order to process large amounts of data efficiently. SOFIE is a very large database application, using a relational database and SQL to manage the data. It is built on standard Oracle database technology, using 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.

As Oracle is the world market leader both in terms of application platforms and database management systems [23], the subject system is built on technology that is widespread across the world. Furthermore, the SOFIE project was also selected as a case because it is representative of large database applications developed by the tax department and other public administrations, with a typically long lifespan. These systems process large amounts of data, leading to an extremely large number of potential test scenarios, which makes testing challenging and hard to scale manually.

From the above, we can see that the subject system selected is representative of many large database applications in numerous environments with respect to technology, size and complexity, and a large user base.

4.2.2. Regression test process

This section is meant to cover relevant aspects of what Petersen and Wohlin refers to as *Processes*, *People* and *Practices, Tools and Techniques*. The primary aim in this section is to describe the regression test process of our subject system. For a general reference on regression testing we refer the reader to a recent survey on the topic conducted by Yoo and Harman [24].

The releases of SOFIE follow a typical waterfall development process, and the regression test effort referred to in this case study is targeted towards system

level testing, which is often the most critical and difficult testing phase. A release matches a set of work tasks, including defects that arose from faults detected in previous test campaigns or from production, or newly specified functionalities. The work tasks are specified, estimated, prioritized and split into releases. The development team develops the changes, performs unit testing, and does the handover to the system testing team. As mentioned in Section 2.1, the regression testing of the batch programs, which are in focus here, is driven by the tool DART [5]. Such a regression test has mainly three prerequisites, namely a test configuration, a test model, and test data. The regression tests are usually carried out by a technical tester, and if he does not possess enough knowledge about the functional domain under test, he will involve a functional tester to conduct the deviation analysis.

For a particular functional domain under test, the tester defines a test configuration, which is the specification of which database tables and columns to monitor during the test execution. Additionally, the tester specifies how to logically group the database manipulations that were captured during the test execution. The typical case in our context is that one taxpayer equals one test case. Consequently, the database manipulations captured during test executions are grouped per taxpayer and the test results are presented per taxpayer (test case). More details can be specified, but what to monitor, and how to group the data into test cases are the most important ones. Test configurations are usually defined once and re-used for subsequent regression test campaigns.

Recall from Section 3 that we use classification tree models to select test cases. Thus, modeling the functional domain under test is an important part of the regression test procedure. To ensure the quality of the models, the modeling has been a combined effort of the testers and domain experts. The domain experts possess in-depth knowledge about both the functional domain and the expected system specification and are thus best suited to make models that capture reality, whereas the tester has knowledge of the modeling tool (CTE XL) and facilitates modeling sessions. In our environment, we have mainly depended on domain experts as system specifications tend to be out of date or not at the right level of detail to use as modeling input. Dalal et al. [25] emphasize the importance of domain knowledge as a requirement for building proper models, as they are able to provide detailed inputs to the modeling process. The output of the modeling process is a classification tree model used as basis for test selection and regression testing.

The tester also prepares a set of test data, either generated synthetically based on the scheme presented in Section 3.3 or based on production data. When the test configuration is defined, the model is created and the test data is ready, the regression tests are executed and analyzed. A technical tester sets up the test and executes it, while potentially bringing in a functional tester to assist the analysis of the test results, i.e. the regression test deviations between the system versions. The regression tests exercise several parts of the system under test, and the technical tester in charge of the test execution does not necessarily have the detailed knowledge required to analyze the details of the deviations in the various tests.

The Norwegian Tax Department is a governmental institution, and does not compete in a market segment, but rather facilitates the tax processing of all Norwegian taxpayers. A further description of the market facet, as suggested by Petersen and Wohlin, is regarded irrelevant in our case.

4.3. Data collection

To collect data to address RQ1, we prepared test data based on production data. The usual practice when testing SOFIE is to make a copy of the production database and then reuse input files from the production environment. We loaded eight randomly selected files of varying size into the test database, which in total constituted data for approximately 42.000 taxpayers. Each input file may be seen as a production data test suite. The data we chose to use forms the foundation for testing four different parts of the system under test, each having an associated classification tree model. We matched the data against the four models, as explained in Section 3.2, in order to measure partition coverage and the level of redundancy.

Addressing RQ2 entails to analyze the operational profile of the functional domains considered. As this is effort-intensive, we chose to focus on three important functional domains of SOFIE, namely those concerning the tax calculations. Two of the models capture slightly different perspectives on the main tax calculations, whereas the third is concerned with changes in tax calculations, i.e. the taxpayer report a change and the tax is recalculated. For each of these domains, we had to go through the following procedure:

- Analyze the operational profile of the system under test, in relation to the classification tree model. For the models related to the main tax calculation, we analyzed all 4.285 million tax calculations for a given year, by matching them to the partitions in the classification tree. For the model related to the

changes in tax calculations, we analyzed all the tax calculations for the past five years, yielding a combined number of 1.2 million calculations. Since this is highly time consuming, we had to focus on specific years, rather than inspecting all data in the entire database. The analysis was conducted in the manner elaborated in Section 3.2, in which all the operational data was matched against the test models (partitions). We could then determine how many test cases fell under each model partition, and could then compute their relative frequencies and identify the most common scenarios executed in the operation of the system. This is referred to as *the operational profile* in the remainder of the paper.

- Generate combinatorial test suite specifications that ensures pair-wise and three-wise model coverage, according to the built-in algorithms in the modeling tool CTE XL [4].
- Assign weights to the model property equivalence classes according to their frequency of occurrence in the data from the operation profile, which we then used to generate a combinatorial weighed pair-wise test suite specification according to the built-in algorithm in the modeling tool CTE XL [4].
- Compare the combinatorial test suite specifications with the operational profile of the system, to understand how well they align.

Regarding RQ3 and RQ4, we ran actual regression tests in order to compare the fault detection capabilities of each test strategy. The functional domain under consideration was the changes in tax calculations mentioned above. Ideally, we would have liked to run regression tests for several program versions and several different functional domains. However, running alternative test suites over several versions and functional domains on both synthetic test data generated according to various test strategies and production data entailed an effort that was far beyond what was feasible. Thus, we chose to focus on one particular functional domain, that had undergone refactoring in a previous system release, so that we knew it had to contain corrections and regression faults. For this particular release, we ran regression tests for all the test strategies mentioned in RQ3. More specifically, we first generated combinatorial test suite specifications ensuring pair-wise, three-wise, and weighed pair-wise model coverage, and then generated executable test cases for each of them according to the test data generation scheme explained in Section 3.3.

Ideally test techniques should be compared with equal cost, and thus the test suite size should be kept constant across techniques. But whereas we can directly control the test suite size for operational profile testing and production data, the test suite sizes for the combinatorial test strategies are determined by the algorithms generating them. For example, three-wise generation will naturally result in more test case specifications than pair-wise. In our case, the pair-wise and weighed pair-wise, resulted in 139 and 147 test cases, respectively, while three-wise resulted in 814 test. Thus, we chose to first determine the most cost-effective combinatorial technique and then compare it with test suites based on the operational profile and production data, with fixed test suite sizes. We evaluated the cost-effectiveness of the combinatorial tests in terms of the number of distinct faults found, relative to the number of test cases executed and the number of deviations to inspect, which respectively capture the cost of test execution and analysis.

After identifying the most cost-effective combinatorial technique, we prepared test suites based on the operational profile and production data. We set the test suite size equal to the number of test cases in the most cost-effective combinatorial technique to enable a fair comparison across techniques. The operational profile test suite was generated according to the procedure described in Section 3.3. Regarding the production data, we sampled test cases both randomly and based on a *Partition-based selection* from the set of production data available. The partition-based selection, which is explained in more detail in Rogstad et al. [8], selects random partitions in a round-robin manner among all covered partitions in the classification tree model, while selecting random test cases within each partition, until the desired number of test cases is selected. For both the operational and the two production data strategies we repeated the exercise 30 times, in order to account for randomness. Using at least 30 repetitions is a common rule of thumb to ensure sufficiently robust statistical results in cases where each repetition is time consuming [26].

For all test strategies, we executed actual regression tests on the selected release of the subject system and analyzed deviations, in order to investigate their fault detection capabilities.

In the guidelines of Runeson and Höst, our data collection corresponds to a second degree data collection as we are directly in control of the collection of our raw data through the execution and analysis of the regression test cases on the subject system. We should also note that the data for each research question was col-

lected independently. However, the functional domain and the model referred to as Model D in Section 4.5 are part of the investigation regarding all four research questions.

4.4. Evaluation methodology

Regarding RQ1, we will analyze model coverage in two complementary ways, namely partition and leave coverage. We will report the maximum number of partitions (all combinations) of each model, along with the total number of leaf nodes, and in turn report the number of partitions and leaf nodes covered by each file of test data for each of the four subject models. The level of redundancy in the test data will be measured by the number of instances that cover the same partition. The formula for redundancy is given by equation 1, where $R(T)$ denotes the level of redundancy, R, for a test set T, $P(T)$ denotes the partitions P covered by test set T, and $|T|$ and $|P(T)|$ denotes the cardinality of the sets.

$$\mathrm{R(T)} = \frac{|\mathrm{T}| - |\mathrm{P(T)}|}{|\mathrm{T}|} \tag{1}$$

RQ2 is about assessing how representative combinatorial test suite specifications are of the system usage. Thus, we compare the test suite specifications of the combinatorial test strategies, with the test suite specification based on the operational profile. Recall that we match operational data against the test models (partitions) to determine how many test cases fall under each model partition, and can then compute their relative frequencies and identify the most common scenarios executed during the operation of the system. For each combinatorial technique, pair-wise, three-wise and weighed pair-wise, we compare their partition coverage against that of the test suite specification based on the operational profile. Two test case specifications are considered to be equal if they share the same combination of equivalence classes (partition). To measure how representative the combinatorial test suite specifications are of the system usage, we report the number of overlapping partitions with OP testing along with the cumulative usage probability of all these overlapping partitions.

RQ3 is about assessing the fault-revealing power of different regression test strategies, both in terms of the number of faults detected and the importance of the faults. In the regression test campaigns, deviations are both due to regression faults and correct changes. In our evaluation, we have chosen to focus on faults, as finding faults has been the primary aim of the regression testing effort. The correctness of changes are usually verified in separate tests. We measure the fault detection power of a test strategy by assessing the number of distinct regression faults revealed relative to the number of test cases in the test suite executed. We also want to favor test strategies that reveal important faults, which in our context means faults that affect many taxpayers, i.e. the most common scenarios of the system operation. So for each fault, we measure the relative likelihood of a test case causing the fault being executed based on data used for the operational profile analysis, i.e. how many taxpayers it would potentially affect. But a limitation is that these results says nothing about the consequences the fault would have. Nevertheless, such likelihood will hereby be referred to as the *importance* of a fault. Recall from above that the operational profile data for the functional domain covered by our regression tests concerned 1.2 million taxpayers over a period of five years. So, when we report an importance of 2,377 (0.19%), it means that over the past five years, the scenario causing the fault has been executed 2,377 times in production, which is 0.19% of the 1.2 million executions that were analyzed.

When comparing the combinatorial techniques, the cost is not constant, as the test suite sizes vary across techniques. Thus, we assess their cost-effectiveness by measuring the number of distinct faults relative to the number of test cases executed and the number of deviations resulting from the tests, along with the cumulative importance of the faults revealed. When we then compare the most cost-effective combinatorial technique with other techniques (OP and PD), we keep the number of test cases constant across techniques, which renders the number of faults and their importance directly comparable.

RQ4 is meant to explore the possibility of combining a combinatorial technique with OP or PD for improved fault detection power. In that respect, we assess to which extent the faults from different techniques overlap. Complementary techniques should be as non-overlapping as possible in terms of the faults they detect, to justify the extra cost of combining them. Then we combine techniques that find complementary regression faults and evaluate the resulting cost-effectiveness.

4.5. Results

In this section we will report and discuss the results of our case study, for each of the research questions defined above.

4.5.1. Analysis of production data (RQ1)

Tables 1, 2, 3 and 4 all follow the same structure and present the results from matching production data used

for testing purposes against the four classification tree models associated with different functional domains. Column 1 states the source of the test cases (file, i.e. test suite), column 2 shows the number of test cases in the test data set, column 3 shows the number of model partitions covered, column 4 shows the number of leaf nodes covered, and column 5 shows the level of redundancy in the test data set. The total number of possible partitions and the total number of leaf nodes contained in the models are given in the caption of each table.

Table 1: Production data matched against test model A, which has 31 leaf nodes and 49,000 possible partitions.

Input	# Test cases	# Partitions covered	# Leaf nodes covered	Redundancy level
File 1	22,894	170	31	99.3%
File 2	10,068	118	31	98.8%
File 3	3,886	89	30	97.7%
File 4	2,310	74	30	96.8%
File 5	1,187	60	29	95.0%
File 6	687	39	29	94.3%
File 7	518	44	28	91.5%
File 8	450	54	28	88.0%
Entire set	42,000	233	31	99.5%

Table 2: Production data matched against test model B, which has 39 leaf nodes and 227,000 possible partitions.

Input	# Test cases	# Partitions covered	# Leaf nodes covered	Redundancy level
File 1	22,894	147	36	99.3%
File 2	10,068	142	36	98.6%
File 3	3,886	94	35	97.6%
File 4	2,310	81	36	96.5%
File 5	1,187	67	32	94.4%
File 6	687	57	33	91.7%
File 7	518	54	34	89.6%
File 8	450	54	33	88.0%
Entire set	42,000	279	36	99.3%

Table 3: Production data matched against test model C, which has 48 leaf nodes and 7,667,000 possible partitions.

Input	# Test cases	# Partitions covered	# Leaf nodes covered	Redundancy level
File 1	22,894	132	35	99.4%
File 2	10,068	118	36	98.8%
File 3	3,886	83	35	97.9%
File 4	2,310	66	34	97.1%
File 5	1,187	60	33	95.0%
File 6	687	50	33	92.7%
File 7	518	51	32	90.1%
File 8	450	46	33	89.8%
Entire set	42,000	204	36	99.5%

Distribution and redundancy of test cases across model partitions. The key thing to note is that, in general, the relative partition coverage in comparison to the number of test cases is very low, which implies that the level of redundancy is very high. For example, the largest set of test data (*File 1*), contains 22,894 test cases, but covers only 170, 147, 132, and 601 partitions in each of the four classification tree models, respectively. Given the formula for redundancy from Section 4.4, this yields a redundancy level between 97.4% and 99.4%, which is extremely high. In our context, a test run on 22,894 test cases would be considered to be a fairly large test. Nevertheless, when systematically matched against classification tree models covering the most important test properties of the functional domains under test, partition coverage is very low. One could argue that a model will always be a simplification of the real world, and not capture all the properties of the test data, thus making the level of redundancy look larger than it actually is. However, these models were prepared by testers and functional experts who know these functional domains very well. Consequently, and although being simplifications as all models are, they are thorough and representative of what can be expected in practice, thus suggesting the number of redundant tests is actually very high.

Table 4: Production data matched against test model D, which has 38 leaf nodes and 746,000 possible partitions.

Input	# Test cases	# Partitions covered	# Leaf nodes covered	Redundancy level
File 1	22,894	601	38	97.4%
File 2	10,068	1,083	38	89.2%
File 3	3,886	832	38	78.6%
File 4	2,310	647	38	72.0%
File 5	1,187	432	38	63.6%
File 6	687	288	38	58.1%
File 7	518	263	38	49.2%
File 8	450	253	38	43.8%
Entire set	42,000	2.762	38	93.4%

As the data shows, partition coverage is significantly higher for Model D than for the three other models. A plausible explanation is that this functional domain benefits from having production data more widely distributed across different years of income than the others. That is, the more data from past fiscal years, the higher the variety of test scenarios. Model D is related to changes in tax calculation. Each tax change adds a level of complexity, and the older the data is, the more likely it is to observe several tax changes. In this case, we loaded data for five different years, which had a positive effect on the dispersion of data for Model D, but did not affect data dispersion for Models A, B, and C.

Overall partition coverage. In the caption of each of the result tables, the total number of valid partitions is given for each of the four models. In general, testing all partitions is not a practical and realistic alternative.

Table 5: Combinatorial test suite specifications matched against the operational profile of the system under test.

Model	Pair-wise			Three-wise			Weighed pair-wise		
	# partitions	# overlapping partitions	% represented	# partitions	# overlapping partitions	% represented	# partitions	# overlapping partitions	% represented
Model D	139	2	0.08 %	814	5	0.13 %	147	5	24.4 %
Model E	67	0	0 %	477	0	0 %	75	9	63.6 %
Model F	52	0	0 %	357	0	0 %	76	7	64.3 %

Thus, measuring coverage against the total number of partitions may not be that relevant. Nevertheless, test data sets cover between 0.0027% and 0.48% of all possible partitions for the four models, e.g. 279 out of 227,000 partitions for Model B. These percentages can be considered to be extremely low.

Leaf node coverage. In the caption of each of the result tables, the total number of leaf nodes is given for each of the four models. Recall from Section 3.2 that a leaf node is an equivalence class at the bottom level of the model. When assessing the entire set of test data combined (42.000 test cases), we see that Model A (31) and Model D (38) cover all leaf nodes, whereas this is not the case for Model B (36 /39) and Model C (36 /48). But also for Model A, not all leaf nodes are covered for the smaller sets of test data (files). So even when running the test data in all files (e.g. for Model C) or a subset of the data in one file (e.g. for Model A), we are not guaranteed to cover all the leaf nodes at least once. The premise of combinatorial testing is that pair-wise or even three-wise coverage of the leaf nodes is desirable. But in many cases, when relying on production data for testing, not even leaf node coverage (also called "each choice" coverage) is ensured. Such "holes" in the production data may be due to seasonal changes or simply rare scenarios that are less frequent and thus fall outside the selected test data set.

Summary of RQ1. Our results suggest that, when relying on production data for testing, the level of redundancy in the test data is high and partition coverage is low. For two out of four test models, not all leaf nodes were covered even when considering the entire set of data combined (all files, 42,000 test cases), and the overall partition coverage across the four models varied from 0.0027% to 0.48%, which is extremely low. For example, the partition containing the highest number of instances for the four models included 25,614, 28,213, 23,299 and 10,931 instances, respectively. This means that, for the case under study, between 26% and 67% of the test data is contained in one partition, i.e. test data representing similar test scenarios. This implies that instances of the test data should be carefully selected when the tests are based on production data. Otherwise, large numbers of redundant tests will likely be executed. Furthermore, even when selecting test data systematically, not all leaf nodes in the model are guaranteed to be covered by the entire test set for all the models. Thus, we can expect that synthesized data will be needed in many cases.

4.5.2. Representativeness of combinatorial test suite specifications (RQ2)

Table 5 shows results from comparing the test suite specifications based on the operational profile of the system, with combinatorial test suite specifications for three different classification tree models. The combinatorial test suites were generated according to the three criteria described in Section 3.3, namely pair-wise, three-wise and weighed pair-wise. Column 1 indicates the model under consideration, columns 2 to 4 show comparisons with pair-wise test suites, columns 5 to 7 show comparisons with three-wise test suites, and columns 8 to 10 show comparisons with weighed pair-wise test suites. For each of the comparisons, we show the number of partitions in the combinatorial test suite specifications in the first column, the number of overlapping partitions with the test suite specification based on the operational profile in the second column, and the cumulative probability of the overlapping partitions in the third column, i.e. how many test cases in the operational profile those overlapping partitions represent.

The first thing to note is that, for Model E and Model F, there is no overlap whatsoever between the pair-wise and three-wise test suite specifications and the operational profile. For Model D, there are only 2 and partitions out of 139 and 814 test case specifications, respectively, that overlap between the pair-wise and three-wise test suite specifications and the operational profile, which is still very few. In practice, that means that executing the pair-wise or three-wise test suites for Models E and F, would not execute a single test case with an exact combination of equivalence classes (partition) equal to one from the real operation of the system. That does not necessarily mean that the test cases will not reveal faults relevant for the operation of the system, as faults can occur due to combinations of a subset of equivalence classes, that in fact are present in the operation of the system. However, it certainly offers a plausible explanation on why many of the faults found using these

techniques were rejected by engineers (chosen not to be corrected), as discussed in Section 2.2.2.

As expected, there is greater overlap when comparing the weighed pair-wise test suite specifications with the operational profile. However, there are still relatively few test case specifications overlapping, the exact numbers being 5 out of 147, 9 out of 75 and 6 out of 76. In terms of the cumulative probability, as shown in the last column of each comparison (columns 4, 7, and 10), the numbers are significantly higher for the weighed pair-wise approach. This is also as one would expect, namely that the weighed approach is better aligned with the operational profile. However, the main reason why the numbers are that much higher, is that the weighed pair-wise approach always starts by generating the most likely test case specification, i.e. the test case specification that contains the most probable combination of equivalence classes. Recall the results from RQ1, showing that the most frequent (likely) partitions typically contain a large percentage of the test data. Thus, by always including the most frequent partition, it will always represent a fair amount of the operational profile data. In general though, most of the test case specifications generated by the weighed pair-wise approach are not a part of the real operation of the system.

Summary of RQ2. In general, a test case specification derived from a combinatorial technique provides little to no overlap with a test suite specification generated from the operational profile. Although the test strategies have different aims, the results are surprising, as we would have expected that the test case specifications bore some relation to the operational profile, at least among the less frequent scenarios. A plausible explanation is that the models tend to be complex, i.e. contain many nodes (58, 196, and 136, respectively) and for many model properties one equivalence class usually has a much higher probability of occurrence than the others. 2W and 3W test suites treat all equivalence classes equally, and are then likely to combine several equivalence classes that are highly unlikely to be combined in practice. It does not take many combinations of less probable equivalence classes before obtaining a highly unlikely test case specification. Furthermore, when the models are large, the chances are high that unlikely combinations are present in most test case specifications. The weighed pair-wise approach helped increase the number of test case specifications overlapping with the operational profile ones and their associated cumulative probability. However, the number of overlapping test case specifications is still very low and a large portion of the cumulative probability is obtained through the coverage of the most likely model partition (recall the results from RQ1). Does the lack of representativeness of combinatorial testing, when compared to the operational profile, render it useless in our context? The results presented in the next section, regarding RQ3 and RQ4, investigate what this means in practice, in terms of revealing regression faults.

Table 6: The regression tests for the combinatorial test strategies.

Test technique	# Test cases	# Deviations	# Distinct faults
2W	139	62	12
W2W	147	54	12
3W	814	396	13

4.5.3. Fault revealing power (RQ3)

Recall from Section 4.3 that we will first identify the most cost-effective combinatorial technique, compare it with OP and PD at constant cost (RQ3), and then try to figure out a cost-effective way to combine techniques (RQ4).

Combinatorial test strategies. Table 6 contains some key figures of the regression tests for the combinatorial techniques. Column 1 specifies the regression test strategy considered, column 2 shows the number of test cases executed, column 3 shows the number of deviations resulting from regression testing, and column 4 shows the number of distinct regression faults found in regression testing. As the table shows, 3W identifies the highest number of distinct faults, but the difference between the three strategies is small. Table 7 shows the details of the test results per regression test suite, namely which faults that were revealed in which regression test strategy, if any. Additionally, the last column, "Importance", shows the number and percentage of tax payers each fault would affect, as explained in Section 4.4.

When investigating the combinatorial techniques (pair-wise, three-wise and weighed pair-wise) in isolation, we can see from Table 6 and Table 7 that there is very little difference in terms of the faults they capture. In fact their fault detection patterns are almost identical. 11 of the 12 faults detected by 2W and W2W are the same, whereas 3W subsumes the faults identified by 2W and W2W, with a total of 13 faults. Despite substantial additional cost, the gain of running three-wise tests is relatively limited compared to running pair-wise or weighed pair-wise. It reveals one additional fault, but at the cost of executing 814 test cases versus 139 and 147, respectively. Moreover, and more important in practice, as executing 814 tests may not be a major hurdle, 3W results in 396 regression test deviations, as

Table 7: The faults revealed by the combinatorial techniques in the subject regression test for the case study.

Fault	Test strategy			Importance
	2W	W2W	3W	
Fault 1	1	1	1	278 (0.02%)
Fault 2	1	1	1	1,022 (0.08%)
Fault 3	1	1	1	53 (< 0.01%)
Fault 7	1	1	1	12,022 (0.97%)
Fault 8	1	1	1	49 (< 0.01%)
Fault 9	1	1	1	192 (0.02%)
Fault 10	1	1	1	1,015 (0.08%)
Fault 11	1	1	1	53 (< 0.01%)
Fault 12	1	1	1	34 (< 0.01%)
Fault 16	1	1	1	49 (< 0.01%)
Fault 17	1	1	1	106 (0.01%)
Fault 18	0	1	1	25 (< 0.01%)
Fault 19	1	0	1	16 (< 0.01%)
Sum faults	12	12	13	

opposed to 62 and 54 for 2W and W2W, respectively. Inspecting deviations is the most labour intensive activity in regression testing, so having to consider 396 deviations for inspection rather than 62 or 54, for revealing one more fault, may not be cost effective. Neither of the additional faults (Fault 18 and 19) detected by 3W, when compared to either 2W or W2W, are among the most important ones. Thus, a pair-wise strategy should be preferred over three-wise, since the gain in fault detection is marginal, whereas the effort spent could be considerably higher. Among the pair-wise strategies, it seems most natural to favor the regular pair-wise strategy. They revealed the exact same number of faults and apart from one exclusive fault each, they captured the exact same faults as well. They also share similar cost, in terms of the number of test cases to execute (139 versus 147) and the number of deviations to inspect (62 versus 54). However, setting up a weighed pair-wise test suite requires more effort, as weights have to be determined and assigned into the model. Thus, among the combinatorial techniques based on synthetically generated test data, our results suggest that 2W is the most cost-effective strategy in our context. In terms of importance, we see that many of the faults detected by the combinatorial test strategies have a low probability of occurring during the operation of the system. Seven faults represent less than 0.01% of the cases over the past five years of historic data, whereas one fault (Fault 7) stands out as more important than the others.

Operational profile and production data. Having identified 2W as the most cost-effective alternative among the combinatorial techniques, we can then run a comparison between 2W, OP and PD with equal size test suites. The number of test cases was set to 139, and test suites for OP and PD were generated accordingly, as described in Section 4.3. Regarding the two different selection strategies used for production data, we have labeled them random production data (RPD) and partition-based production data (PPD), while using the label PD when referring to the production data in general terms.

Table 8 shows the number of deviations and distinct faults resulting from the different test strategies, reporting averages for OP and PD due to their random nature. 2W detects the highest number of regression faults (12), whereas PD fares better than OP (3.27 and 7.53 faults versus 1.73 faults) with PPD being the most efficient way to sample production data among the two (7.53 versus 3.27 faults). So if the number of distinct faults was the only evaluation criteria, 2W would be superior to the others when compared holding execution cost (test suite size) constant. However, let us consider the distribution of faults and their importance as well. Table 9 shows the details of the test results per regression test suite, namely which faults were revealed by which regression test technique, along with the importance of the faults. For OP and PD, which are partly randomized techniques, the frequency of fault detection is shown.

Table 8: The regression tests for the different test strategies.

Test strategy	# Test cases	# Deviations	# Distinct faults
2W	139	62	12
OP	139	2.04	1.73
RPD	139	16.42	3.27
PPD	139	31.73	7.53

Let us consider OP first. Intuitively, we would expect the OP test suite to discover more important faults than the combinatorial test suites. And in fact, the three unique faults that can be revealed by OP (fault 2, 7, and 10) are the most important ones among the faults captured with synthesized data, representing 0.08% to 0.97% of the cases in the system operation. However, OP does not detect them with a high degree of certainty (frequency $\ll 1$) and these faults are not uniquely identified by OP. PD can capture two of them (fault 2 and 7), while 2W captures all three with 100% certainty. Our initial skepticism towards the combinatorial techniques, and thus the motivation for investigating operational profile testing, was related to their ability to detect important faults. However, the data presented in Table 9, when comparing 2W and OP, indicate that the

Table 9: The faults revealed by all test strategies in the subject regression test for the case study.

Fault	Test strategy				Importance
	2W	OP	RPD	PPD	
Fault 1	1	0	0	0	278 (0.02%)
Fault 2	1	0.47	0.52	0.99	1,022 (0.08%)
Fault 3	1	0	0	0	53 (< 0.01%)
Fault 4	0	0	0.78	0.89	28,883 (2.32%)
Fault 5	0	0	0.24	0.76	2,377 (0.19%)
Fault 6	0	0	0.14	0.67	82 (0.01%)
Fault 7	1	0.83	0.14	0.71	12,022 (0.97%)
Fault 8	1	0	0	0	49 (< 0.01%)
Fault 9	1	0	0	0	192 (0.02%)
Fault 10	1	0.43	0	0	1,015 (0.08%)
Fault 11	1	0	0	0	53 (< 0.01%)
Fault 12	1	0	0	0	34 (< 0.01%)
Fault 13	0	0	0.48	0.98	3,494 (0.28%)
Fault 14	0	0	0.15	0.66	489 (0.04%)
Fault 15	0	0	0.41	0.97	12,893 (1,04%)
Fault 16	1	0	0	0	49 (< 0.01%)
Fault 17	1	0	0	0	106 (0.01%)
Fault 18	0	0	0	0	25 (< 0.01%)
Fault 19	1	0	0	0	16 (< 0.01%)
Fault 20	0	0	0.15	0.67	99 (0.01%)
Fault 21	0	0	0.13	0.22	57 (< 0.01)
Fault 22	0	0	0.14	0.22	233 (0.02%)
Nr of faults	12	3	11	11	

combinatorial techniques can not only find many rare faults, but they also detect the most important ones with a perfect, much higher degree of certainty than OP.

So despite the fact that the partitions covered by combinatorial test suites have low probabilities of occurrence, and thus are rarely or never executed in the operation of the system (as shown in RQ2), they still capture faults that are executed by more common executions. 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, combinatorial test case specifications seem to have the ability to detect important faults. This supports common claims in the literature regarding combinatorial testing when motivating the use of pair-wise and three-wise test suites, namely that faults are often found in the interaction of few input values [18, 27]. According to the results presented, there is no empirical support to suggest that our original skepticism towards combinatorial testing's capability to find important faults was justified. Though they do indeed capture a lot of faults with a low probability of occurrence during system operation, they also capture the higher probability faults than can be detected using operational profile testing and with a much higher degree of certainty. Thus, the fact is that 2W is more likely to capture all the faults covered by the OP test suite and many more, with fewer test cases, and is thus a much more cost-effective strategy.

What does seem to capture additional relevant faults, when compared to combinatorial testing, is production data. Contrary to the combinatorial test strategies, that were almost identical in terms of their fault detection capabilities, the faults that can be detected by the regression test relying on production data are almost mutually exclusive to those detected by the combinatorial tests. Only two out of 11 faults overlap with the faults from the combinatorial tests, whereas the remaining faults are unique to PD testing. This indicates that regression testing based on production data is very complementary to the combinatorial test strategies. However, an effective selection strategy is important since, as shown when addressing RQ1, the production data contains substantial redundancy. Selecting test cases while maximizing partition coverage of the model (as with PPD) is much more cost-effective than RPD, with more than twice the number of detectable faults.

In terms of important faults, PD also fares well. In fact, if we order the list of faults by their importance, six of the 11 faults (Fault 4, Fault 15, Fault 7, Fault 13, Fault 5 and Fault 2) identified by the production data come out on top, four of them being uniquely revealed by the PD test suite. In general, the data clearly suggest that the test suite based on production data identifies more important faults than those executed on synthetic test data. The question then is, why were those faults not discovered using the combinatorial test suites or the OP test suite?

The synthetically generated test data is derived from a test model. While being flexible, i.e. it can generate any combination of test data from the model for any given test, the test data is constrained by the model, or more specifically the imagination and expertise of the people defining the model. A detailed analysis show that for each of the faults that are solely detected by PD, the underlying cause for triggering the fault is not explicitly captured by the model. For example, there was an issue with one particular type of credit transaction for tax payment that the model did not capture. Hence, the synthetic test data generated from the model, as in the case of both the combinatorial test suites and the OP test suite, did not cover that particular fault, whereas testing from production data did. In turn, this shows that modeling a large input domain is hard and that, even with domain experts involved, important properties will be missing from the model. Thus, not using production data to support regression testing could lead to more regression faults being deployed, as the coverage of the

synthetic test data derived from a model is not a strong guarantee of the detection of important faults.

When trying to determine the best test strategy overall, 2W and PPD are the ones that stand out. 2W being the most cost-effective among the combinatorial test strategies, while PPD is a more efficient sampling strategy than RPD for tests based on production data. In terms of the number of faults detected, 2W is far better than PPD with 12 faults detected versus an average of 7.53. In general though, PPD detects more important faults than 2W. As expected from a mature, deployed system, none of the faults found in these regression tests are associated with high probabilities of occurrence in practice. But Fault 4, 15, and 7 represent a fair portion of the system executions, with 2.32%, 1.04% and 0.97%, respectively. PPD finds all these three faults in most executions (88.9%, 97.4% and 70.7%, respectively), whereas 2W detects Fault 7, but not Faults 4 and 15. In the other end of the scale, there are seven faults representing less than 0.01% of the system executions. Six of those were found by 2W, while one of them was found by PPD. Among the 11 faults of intermediate importance, PPD finds the three most important ones (Faults 13, 5, and 2), whereas 2W finds the third and fourth most important ones (Faults 2 and 10). Since 2W finds more faults than PPD, whereas PPD tends to reveal more important faults, determining the best approach to regression testing is then not clear-cut: It is a matter of deciding whether you want to find a larger number of faults with mostly low probabilities of occurrence during operation, as opposed to finding less faults with higher such probabilities.

Summary of RQ3. Among all the test strategies, 3W detected the most regression faults, but at an additional cost of many more test cases executed and far more deviations to inspect. 2W and W2W showed almost identical results, and captured only one fault less than 3W with significantly reduced test suite sizes (139 and 147 versus 814 test cases). And because 2W is a more straightforward technique than W2W and features a smaller test suite, 2W proved to be the most cost-effective approach among the combinatorial ones. OP could only find few faults (3), but those three faults were the most important ones among the faults detected with synthesized test data. However, the same faults were also captured by all the combinatorial techniques, and with perfect certainty since these strategies are deterministic, thus suggesting they are more effective at capturing important faults as well as much less likely ones. The PD techniques did not reveal as many faults as 2W, with an average of 3.27 for RPD and 7.53 for PPD. In general though, PD revealed more important faults than 2W, thus implying that synthesized test data could let important regression faults slip through production. Whether to opt for 2W or PPD if having to choose between the two is hard to decide. 2W detects more faults, whereas the faults detected by PPD are on average more important.

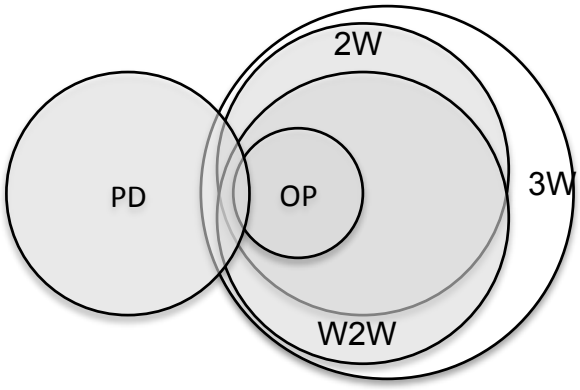

Figure 4: A Venn diagram showing the overlap of deviations between different test strategies.

4.5.4. Combining techniques (RQ4)

Based on the results above, Figure 4 presents a Venn diagram that depicts, in a qualitative way, the overlap of detected faults among the different strategies. The diagram shows that 3W subsumes 2W and W2W, which in turn subsumes OP, while there is little overlap between those four and PD. The fact that PD stands out from the others, in terms of the type of faults it detects, suggests that it should be included as a part of a comprehensive test strategy. The fact that it also revealed the most important faults strengthens its position even more. As the regression faults of OP are subsumed by 2W, it would not provide any additional value to combine the two techniques. Thus, combining 2W with PPD seems to be a natural choice in order to run effective regression tests.

Figure 5 shows the number of faults detected per number of test cases executed, whereas Figure 6 shows the number of faults detected per number of regression test deviations, i.e. the number of regression test cases resulting in a deviation. Although our focus is on the combination of techniques, we have also included all the test strategies evaluated in RQ3 in the graphs to visualize the differences in fault detection capabilities relative to the number of test cases executed and the number of deviations to inspect. In addition to each individual test strategy, we show the results from combining 2W (the

most cost-effective combinatorial approach) with OP, RPD and PPD. For techniques that include an element of randomness (RPD, PPD, OP, 2W+OP, 2W+RPD and 2W+PPD) averages are plotted in the graph. The detailed numbers and basic statistics for both figures can be found in Table A.10 in Appendix A.

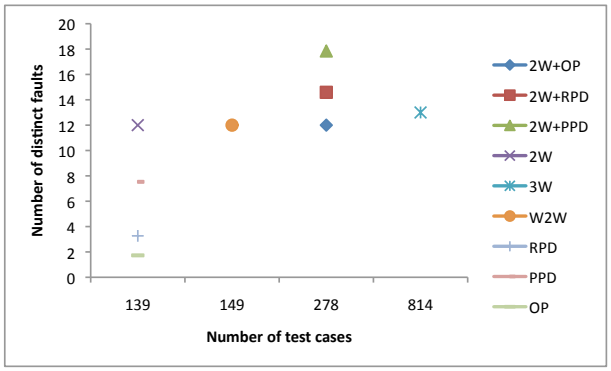

Figure 5: The number of distinct faults found relative to the number of test cases executed.

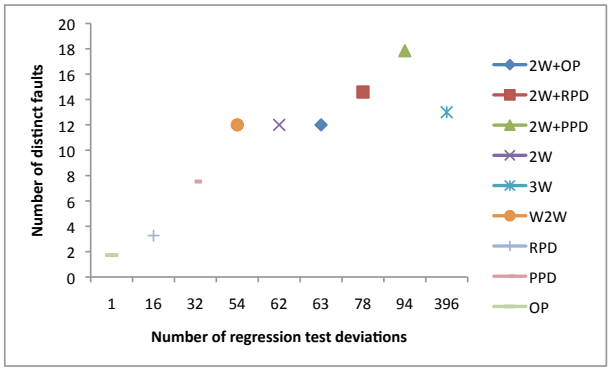

Figure 6: The number of distinct faults found relative to the number of test cases resulting in a deviation.

The graphs confirm what we already anticipated above based on the Venn diagram, namely that combining 2W and OP does not fare any better than 2W alone, since 2W subsumes the faults of OP. It just adds extra costs without any increase in fault detection. Combining 2W with PD on the other hand offers an improved fault detection rate. Whereas 2W and PPD individually detect 12 and 7.53 faults, respectively, they detect 17.85 faults on average when combined, which is a significant improvement. The cost of test execution also increases from 139 to 278 test cases, while the number of deviations increases from 62 to 94. Remember that 3W, which can be seen as an extension of 2W revealed 13 faults from 814 test cases and 396 deviations. In comparison the combination of 2W and PPD reveals an average of 17.85 regression faults with only 278 test cases and 94 deviations, which is far more efficient. Thus, the combination of 2W and PPD seems to be a convincing test strategy in terms of covering a broad range of regression faults at a reasonable cost.

Summary of RQ4. Our working hypothesis was that the combination of a combinatorial test suite and a test suite based on the system operational profile, both based on synthetic test data generated from a classification tree model, would be the most powerful combination. The assumption was that combinatorial tests would capture faults that would only exceptionally occur in practice while the operational profile tests would reveal faults with high probabilities of occurrence. However, as the data clearly shows, the best combination seems to be combinatorial testing and production data, and more specifically the combination of 2W and PPD.

It also turned out that our initial skepticism towards combinatorial test suites, namely that they were not able to detect important faults, did not fully reflect reality. Despite being constituted of test case specifications with a low probability of occurrence in the system operation, they did also capture important faults within the scope of the model they are generated from. Thus, deriving an operational profile test suite, from the same model, did not provide any additional value, as the faults detected by the OP test suite were also captured by the combinatorial test suites. So rather than synthesizing an OP test suite, it turns out it is better to rely on production data as a complement to synthesized combinatorial test data. Production data will by nature align with the operational profile and is complementary to combinatorial test data in terms of fault detection. The main reason seems to be that, despite being developed by domain experts, it is very difficult in practice to devise complete test models (classification trees).

4.6. Recommendations

We have implemented a systematic approach for generating synthetic test data, i.e. the necessary input data in order to execute a test case, based on test case specifications given by a classification tree model (recall Section 3.3). Although this entails an initial setup and implementation cost, once established, it provides us with the ability to generate test data for any given test case specification derived from the model. In turn, this enable us to run regression tests efficiently, as the preparation of test data is automated. The fact that the test data is generated from the test models also ensures a

systematic test approach and predictable model coverage across regression tests. Setting up production data for a regression test requires a few more steps including manual intervention, as input files from the production environment have to be located and transferred to the physical machine of the test database, where the data is anonymized and read into the test database. Additionally, our results suggest that the model coverage of the production data is low and unpredictable, while at the same time such data needs to be carefully selected because of its high level of redundancy from a testing standpoint.

Another practical benefit of synthesized test data is related to regression test execution and analysis. Remember that our regression test approach compares executions of a changed version of the program (delta) against the original version of the program (baseline) in order to identify deviations in database manipulations on a selected set of database tables and columns. Such an approach requires the two test runs to be executed on the same set of test data and the same initial database state. When using production data, we are not able to rerun the test data used in the baseline execution for the delta execution, as the state of our taxpayer data has changed during the baseline execution. Thus, we have to run a "flashback" on the database, which sets the database back to its initial state, before rerunning the test case on the changed program version with the exact same test data and database state. Because of the need for a database flashback, the execution of the baseline is no longer present in the database after the completion of the regression test. The drawback then is that the tester only has the delta execution available in the database when conducting the test analysis. This is not a major obstacle, as all the relevant data for the regression test was captured and stored during the baseline run. However, it is convenient for the tester to have both executions available, in order to directly compare them in the system under test. When using synthesized data, the approach is slightly different, as we generate a new set of test data for each run. The two sets of test data are identical, i.e. share the exact same properties and values, apart from the fact that they are created on separate fictitious taxpayers. This enable us to generate identical sets of test data for each test run (baseline and delta) that are directly comparable, without having to flashback the database.

As described above, there are some practical implications that make synthesized test data better suited for running effective regression tests. Thus, based on that assumption, the ideal scenario would be to run regression tests on a combination of combinatorial test data and operational profile test data, both synthetically generated. However, as the results from RQ3 and RQ4 show, running regression tests on production data turns out to be both efficient and complementary to running regression tests on synthesized combinatorial test data. Not accounting for this result could lead to important faults slipping to production. This result is mostly due to the fact that, even when relying on domain experts, test models are usually incomplete and do not account for all important factors affecting test results.

As regression testing is more effort intensive with production data, we recommend running continual regression tests using the most cost-effective synthesized combinatorial test data (2W) 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 (PPD) 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 synthesized 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.

4.7. Threats to validity

In this section, we discuss the generality and potential threats to validity of out case study, following the classification scheme of Yin [28], as further described for software engineering in the guidelines by Runeson and Höst [21].

4.7.1. Construct validity

Our goal was to assess the use of production data for regression testing and the alternative practice of using synthetic test data generated by following various combinatorial strategies and operational profiles based on classification tree models. Because we chose to use classification tree models to drive the generation of synthetic test data, it was logical to assess the production data on the basis of the same models. While the synthesized data is generated according to a given model coverage level, the production data is not. Thus, we evaluated the model coverage and redundancy level of production data to better understand what to expect when testing with production data. We measured model

coverage in three complementary ways, namely model property, partition, and leave coverage, which all tells us about the coverage of test case specifications. Additionally, we measured the redundancy level of the production data in terms of duplicate test cases based on the classification tree model.

As for the comparison of techniques, we used the standard measure of fault detection rate, but also accounted for fault importance as we wanted to favour strategies that reveal faults likely to occur in operation. We also investigated the combination of test techniques in a way that ensures more complete regression testing.

We thus believe we have used appropriate operational measures to adequately compare test techniques and address our research questions, and do not believe there should be any immediate threats to construct validity.

4.7.2. Internal validity

We ensured that test suites were, when warranted, comparable in terms of cost across techniques. Therefore, we do not see any threats as to whether the observed relationship between the treatment (test techniques) and results (e.g., fault revealing power) could be explained by other factors.

4.7.3. External validity

An important question is whether our results, based on regression tests of one large system, are likely to be generalizable to other database applications. Ideally, one would always want to run multiple studies to better ensure the generalizability of the results. However, finding the type of data needed to conduct a similar study is not available in open source systems, and to replicate our study on another real world, large scale system would be a major, multi-year undertaking requiring the collaboration of another development organization. We believe the following factors helps mitigate the threat to external validity of the study:

- Our regression tests are based on information that should be easy to collect from any database application, namely changes in database manipulations between two versions of the system.
- The case study is conducted on a widely used database application and evaluated on real regression tests capturing real regression faults.
- As discussed in Section 4.2, 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 [23], the system is built on technology that is widespread across the world.
- 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 effective test strategies crucial during system evolution.

Another potential threat to external validity is the data collected and used to address RQ1. Such data may be domain specific, that is specific to our context and case study system. The fact that we analyzed data from four different functional domains of the system under test should reduce this threat. Further, having operational scenarios that are vastly more frequent than others is quite common in most systems. Nevertheless, additional studies are warranted to obtain a broader picture of the degree of redundancy to be expected in the production data of most systems.

4.7.4. Reliability

The results are based on an analysis of production data, the collection of operational profile data and comparisons of test techniques across regression tests. The data collection and analysis is clearly described as part of the case study design, and we do not see any potential threats as to how the results should be biased or dependent on the people conducting the study.

5. Related Work

The scope of our related work section includes works on combinatorial testing, database testing and operational profile testing.

Nie and Leung published a survey on combinatorial testing (CT) in 2011 [29]. They analyzed more than 90 key papers within CT research in order to evaluate the current state of the research and make recommendations for the future. Test case generation, i.e. generating small combinatorial test suites of a certain level of coverage, has by far been the most active area of research within CT. More than half of the papers (50) were related to generating combinatorial test suites. Combinatorial testing being an NP-hard problem, requiring approximations to be addressed, this topic has been given a great deal of attention throughout the years. In 2005, Grindal et al. conducted a survey more specifically within this area of CT research, surveying papers on

various combinations strategies for generating test cases [30].

Regarding future research within CT, Nie and Leung propose several directions within different subcategories of CT. We would like to highlight three of them, in which we believe our study provides the most significant contributions. First, the authors emphasize the lack of empirical results in CT and stress the need for more studies and collection of more evidence to fully understand the limitations and strengths of CT. Our study provides empirical results on the application of combinatorial testing on a large system in a real industrial context. Second, Nie and Leung suggest that more research is needed on the application of CT to different levels of testing and different types of software. In our study, combinatorial techniques are applied to functional regression testing of a large industrial database application, a combination not previously covered in CT research. Third, the authors state that more work is required in integrating test case generation tools with other tools to fully automate the test process. In our study, we show how a tool for combinatorial testing (CTE XL) can be integrated with a tool for automated regression testing (DART), to automate functional system testing and regression testing, i.e. partially automated regression tests running automatically generated combinatorial test suites. Grindal et al. stated that further investigations are needed on how to select an appropriate combination strategy for a particular test problem. Although, we do not provide a general procedure on how to select an appropriate combination strategy for a particular test problem, we compare various combinatorial strategies used for functional system level regression testing of a large industrial database application. Our results suggests that pair-wise is the most efficient combinatorial strategy for that particular purpose.

Musa (1993) defined an operational profile (OP) as a quantitative characterization of how a system will be used [31]. Operational profiles are primarily used within software reliability testing to ensure that the most important features of the system under test are tested first. Smidts et al. conducted a survey of software testing relying on operational profiles [7]. Their main focus was to address the characteristics of OP development and their classification and relationships. The authors ended up selecting 19 papers that were carefully analysed.

Smidts et al. (2014) found that several modeling paradigms are used to describe and build an OP, namely trees, state-based representations, probabilistic event flow graphs and sets, with state-based being the most common one. Recall that we build our OP of the input domain based on classification tree models by matching the equivalence classes of each model property against actual historic tax input data over a number of years. Six of the papers surveyed also used tree structures as part of their OP representation [31, 32, 33, 34, 35, 36]. However, the traditional approach within OP development is to model the software usage using a tree-based structure, where each tree node corresponds to elements of the particular usage profile. Each node is then assigned a probability of occurrence, which in turn leads to a combined probability of a sequence of events. Although our modeling approach and probability assignment resemble prior use found in the OP literature, two things are also fundamentally different: Whereas OP testing traditionally builds profiles of system usage in order to ensure reliability for the system end users, we have built profiles of the input domain of the system under test to ensure reliable tax calculations for the taxpayers. Our focus is on testing the automated processes for the tax calculations, not the manual system interactions of the end-users, i.e. taxation officers taking various actions to collect tax money remained to be paid. Thus, we have built OPs of the input domain to understand the importance of various test cases (combinations of inputs) such that we make sure to test the scenarios that could potentially harm large numbers of taxpayers. We have not seen such usage of OPs in other studies.

Within the context of database testing, Chays et al. (2000) noted the lack of uniform methods and testing tools for verifying the correctness of databases, despite their crucial role in many organizations [37]. In the following years Chays and Deng et al. proposed a framework, called AGENDA, for testing database applications [37, 38, 39, 40]. The components of the test framework included a parsing tool that gathered relevant information from the database schema and application, a tool that generated test cases for the application, a tool that checked the resulting database state after operations were performed by the database application, and a tool that assisted the tester in checking the output from the database application. However, the tool set remained at a prototype level, handled only very small database applications and were never scaled up to or evaluated for testing a realistically sized database application.

Another relevant research project within the context of database testing is the SIKOSA project [41, 42, 43]. The authors proposed a framework specifically targeted at regression testing of database testing. They focused on black-box tests, using a capture-and-replay approach. This aligns well with our approach, but while they restricted their work to checking input-output-relations of database applications, we also monitor the

database state, as we target regression testing of batch programs that process large amounts of data and reflect their outputs directly in the database. 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.

When compared to the existing literature, our paper offers interesting and practical results from an industrial setting on a real and representative database application, reporting real faults, with a focus on the practical application of combinatorial testing, operational profiles, and their combination.

6. Conclusion

There are very few studies, carried out in real development contexts, investigating the relative cost-effectiveness of black-box system test strategies, for example based on combinatorial strategies or operational profiles. In this paper, based on a large scale case study, we have assessed the practice of using production data for the regression testing of database applications, a common strategy in many development environments. Furthermore, we have also assessed the alternative strategy of generating synthetic test data according to common strategies. For this purpose, we have used classification tree modeling, which is a well-known black-box specification technique, to model the input domain of the system under test. The synthetic test data is generated on the basis of the specifications derived from these models according to both combinatorial and operational profile strategies.

We presented a practical approach on how to integrate a classification tree modeling tool (CTE XL) for black-box test specifications with a regression test tool for database applications (DART). This enables us to match production data against the models to learn their distribution across model partitions. We use this both to develop operational profiles of the input domains and to select production data for testing in a balanced way from the model partitions that have matching test data. The same model specifications are also used as basis to drive the generation of synthetic test data. Thus, we have automated support for generating test suite specifications from the models, generating synthetic test data from the model specifications, selecting production data for testing according to their distribution across model partitions, developing operational profiles, along with a partially automated process for database regression testing.

We matched production data (for approximately 42.000 taxpayers) that formed the foundation for testing four different parts of our case study system against their associated classification tree models. The results showed that the level of redundancy in the production data varied between 93.4% and 99.5% for the entire set of production data used in the study, which is extremely high. In practice, this means that if we would run tests randomly sampling from this data, more than 93.4% of the test cases would be redundant according to our test models. In particular, a large proportion of the data set is contained in a few partitions, e.g. the partition containing the highest number of instances for each of the four models under study included 25,614, 28,213, 23,299 and 10,931 instances, respectively. The results also showed that the partition coverage of the production data was low, e.g. covering 279 out of 227,000 feasible partitions. The high level of redundancy in the production data suggests that production data should be selected carefully for the purpose of testing, to avoid unnecessary costs entailed by redundant tests. Further, the low level of partition coverage suggests the need for generating synthetic test data to increase the completeness of testing.

During initial regression tests based on synthesized combinatorial test data, i.e. pair-wise and three-wise test suites, we experienced that many of the faults reported were not corrected as they were not deemed relevant for the operation of the system. A comparison of combinatorial test suite specifications against test suite specifications based on the operational profile, confirmed that their respective test suites had little in common, thus suggesting that the former were not representative of actual system usage.

Because of impracticalities pertaining to the use of production data for test automation purposes, we were curious to see whether the combination of combinatorial test suites and test suites following an operational profile, both generated synthetically, would yield effective regression tests in terms of fault detection. Our initial hypothesis was that the operational profile tests would reveal faults with high probabilities of occurrence, while the combinatorial tests would mostly capture faults that would only exceptionally occur in practice. We first compared the combinatorial test strategies, namely pair-wise (2W), three-wise (3W) and pair-wise weighed according to the operational profile (W2W), and concluded that 2W was the most cost-effective. 2W executed far less test cases than 3W (139 versus 814), resulted in far less regression test deviations to inspect (64 versus 396) and found only one less fault (12 versus 13). The results between 2W and W2W were very

similar, but 2W is preferred since it requires less effort to carry out. Then we compared the fault detection rate of pair-wise (2W) against test suites based on the operational profile (OP) and production data (PD), while keeping the number of test cases identical. To our surprise, we found that OP did not find any additional faults when compared to 2W, i.e. 2W subsumes the faults of OP. Though the faults detected by OP were the most important ones among the ones detected on synthesized data, it turned out that 2W also detected these faults, despite the fact that most of the 2W test cases did not match realistic scenarios. This was explained by the fact that most faults require test data to cover only a small subset of model property values to be triggered. Thus 2W is able to detect common faults despite executing test cases that combine input equivalence classes in an unrealistic way. On the other hand, when adopting a smart coverage strategy, PD test data captured many faults that were not detected by 2W tests, which we attributed to the fact that no realistic classification tree model can be expected to capture all relevant test properties. Thus, against initial expectations, the combination of 2W combinatorial test data and production data selected in a systematic manner according to model partition coverage (partition-based selection) turned out to be the most powerful combination for the functional regression testing of database applications.

Appendix A. Appendix

Table A.10 reports detailed data and descriptive statistics from which Figures 5 and 6 were generated.

Table A.10: Descriptive statistics (minimum, median, average, maximum, and standard deviations) for the number of faults detected per number of test cases executed and resulting deviations.

Test suite	# Test cases executed	# Deviations (avg.)	Distinct faults				
			Min	Median	Average	Max	Std. Dev.
2W	139	62	12	12	12	12	0.00
W2W	147	54	12	12	12	12	0.00
3W	814	396	13	13	13	13	0.00
OP	139	2.04	0	2	1.73	3	0.68
RPD	139	16.42	0	3	3.27	8	1.36
PPD	139	31.73	4	8	7.53	11	1.17
2W+OP	278	33	12	12	12	12	0.00
2W+RPD	278	78	12	15	14.59	20	1.24
2W+PPD	278	94	14	18	17.85	20	1.07

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.

References

[1] E. Dustin, Orthogonally speaking: Deriving a suitable set of test cases, The software testing and quality engineering magazine (STQE) 2001 (2002).

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

[3] M. Grochtmann, K. Grimm, Classification trees for partition testing, Software Testing, Verification and Reliability 3 (1993) 63–82.

[4] E. Lehmann, J. 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).

[5] E. Rogstad, L. Briand, E. Arisholm, R. Dalberg, M. Rynning, Industrial experiences with automated regression testing of a legacy database application, in: 27th IEEE International Conference on Software Maintenance (ICSM), pp. 362–371.

[6] T. J. Ostrand, M. J. Balcer, The category-partition method for specifying and generating functional tests, Communications of the ACM 31 (1988) 676–686.

[7] C. Smidts, C. Mutha, M. Rodríguez, M. J. Gerber, Software testing with an operational profile: Op definition, ACM Comput. Surv. 46 (2014) 39:1–39:39.

[8] E. Rogstad, L. Briand, Test case selection for black-box regression testing of database applications, Information and Software Technology (IST) 31 (2013) 676–686.

[9] E. Rogstad, L. Briand, Clustering deviations for black box regression testing of database applications, Reliability, IEEE Transactions on PP (2015) 1–15.

[10] D. Cohen, S. Dalal, M. L. Fredman, G. Patton, The aetg system: an approach to testing based on combinatorial design, Software Engineering, IEEE Transactions on 23 (1997) 437–444.

[11] A. 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, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2000, pp. 59–74.

[12] L. Xu, B. Xu, C. Nie, H. Chen, H. Yang, A browser compatibility testing method based on combinatorial testing, in: Proceedings of the 2003 International Conference on Web Engineering, ICWE'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 310–313.

[13] P. J. Schroeder, B. Korel, Black-box test reduction using input-output analysis, SIGSOFT Softw. Eng. Notes 25 (2000) 173–177.

[14] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, Constructing test suites for interaction testing, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 38–48.

[15] G. J. Myers, Art of Software Testing, John Wiley & Sons, Inc., New York, NY, USA, 1979.

[16] R. Mandl, Orthogonal latin squares: An application of experiment design to compiler testing, Commun. ACM 28 (1985) 1054–1058.

[17] A. Williams, R. Probert, A practical strategy for testing pairwise coverage of network interfaces, in: Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on, pp. 246–254.

[18] D. Kuhn, M. Reilly, An investigation of the applicability of design of experiments to software testing, in: Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE, pp. 91–95.

[19] P. Schroeder, P. Bolaki, V. 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, pp. 49–59.
[20] P. Kruse, M. Luniak, Automated test case generation using classification trees, Software Quality Professional 13 (2010) 4–12.
[21] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw. Engg. 14 (2009) 131–164.
[22] K. Petersen, C. Wohlin, Context in industrial software engineering research, in: Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, pp. 401–404.
[23] C. Graham, Y. Dharmasthira, C. Eschinger, B. F. Granetto, J. M. Correia, L. F. Wurster, F. D. Silva, T. Eid, R. Contu, F. Biscotti, C. Pang, D. M. Coyle, D. Sommer, M. Cheung, H. H. Swinehart, B. Sood, A. Raina, K. Motoyoshi, Y. Nagashima, J. Zhang, M. Warrilow, S. Deshpande, J.-S. Yim, V. Mladjov, J. Mazzucca, Market share: All software markets, worldwide, 2013, 2014.
[24] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: A survey, Softw. Test. Verif. Reliab. 22 (2012) 67–120.
[25] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, Model-based testing of a highly programmable system, in: Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on, pp. 174–179.
[26] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability 24 (2014) 219–250.
[27] D. R. Kuhn, D. R. Wallace, A. M. Gallo, Jr., Software fault interactions and implications for software testing, IEEE Trans. Softw. Eng. 30 (2004) 418–421.
[28] R. K. Yin, Case study research: Design and methods, Sage, London, 3rd edition, 2003.
[29] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. 43 (2011) 11:1–11:29.
[30] M. Grindal, J. Offutt, S. F. Andler, Combination testing strategies: a survey, Software Testing, Verification and Reliability 15 (2005) 167–199.
[31] J. Musa, Operational profiles in software-reliability engineering, Software, IEEE 10 (1993) 14–32.
[32] L. du Bousquet, F. Ouabdesselam, J. Richier, Expressing and implementing operational profiles for reactive software validation, in: Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on, pp. 222–230.
[33] S. Elbaum, S. Narla, A methodology for operational profile refinement, in: Reliability and Maintainability Symposium, 2001. Proceedings. Annual, pp. 142–149.
[34] J. Musa, The operational profile in software reliability engineering: an overview, in: Software Reliability Engineering, 1992. Proceedings., Third International Symposium on, pp. 140–154.
[35] F. Ouabdesselam, I. Parissis, Constructing operational profiles for synchronous critical software, in: Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on, pp. 286–293.
[36] P. Runeson, C. Wohlin, Constructing operational profiles for synchronous critical software, in: 1st Conference on European International Conference on Software Testing, Analysis and Review (EuroSTAR'93), pp. 309–323.
[37] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, E. J. Weber, A framework for testing database applications, SIGSOFT Softw. Eng. Notes 25 (2000) 147–157.
[38] D. Chays, Y. Deng, Demonstration of agenda tool set for testing relational database applications, in: Software Engineering, 2003. Proceedings. 25th International Conference on, pp. 802–803.
[39] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, E. J. Weyuker, An agenda for testing relational database applications: Research articles, Softw. Test. Verif. Reliab. 14 (2004) 17–44.
[40] Y. Deng, P. Frankl, D. Chays, Testing database transactions with agenda, in: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 78–87.
[41] F. Haftmann, D. Kossmann, E. Kreutz, Efficient regression tests for database applications, in: In Conference on Innovative Data Systems Research (CIDR, pp. 95–106.
[42] C. Binnig, D. Kossmann, E. Lo, Testing database applications, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 739–741.
[43] F. Haftmann, D. Kossmann, E. Lo, A framework for efficient regression tests on database applications, The VLDB Journal 16 (2007) 145–164.