Industrial Experiences with Automated Regression Testing of a Legacy Database Application

Erik Rogstad, Lionel Briand
Simula Research Laboratory, Lysaker, Norway
University of Oslo, Dept. of Informatics, Oslo, Norway
erik.rogstad@simula.no, lionel.briand@simula.no

Ronny Dalberg, Marianne Rynning
The Norwegian Tax Department, Oslo, Norway
ronny.dalberg@skatteetaten.no,
marianne.rynning@skatteetaten.no

Erik Arisholm
Testify AS, Oslo, Norway
University of Oslo, Department of Informatics, Oslo, Norway
erik.arisholm@testify.no

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

Keywords: regression testing; legacy database applications; industrial context;

I. INTRODUCTION

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

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

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

The main contributions of this paper are:

- A practical approach and tool (DART) for regression testing of database applications, with a focus on generating and prioritizing black-box test cases, automatically identifying potential regression faults and then prioritizing their inspections for early fault detection.
- Application and evaluation of DART for business critical batch jobs in a legacy database application in an industrial setting.

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

II. Testing requirements for the SOFIE system

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

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

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

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

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

III. Problem definition and related work

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

- A batch job consists of a large number of tightly integrated set of operations, which makes it hard to control the job during test. A batch job can only be started, without further mechanisms of control. Then it runs to completion, typically in multiple, parallel job streams. Thus, you can control the input of the batch, and check the end result of it, but what happens in between is difficult to observe, and even more difficult to control.
- For the very same reasons it is very hard to build an automated test oracle (predicting the “expected result”) for the system under test.
- Given the amount of batch jobs in the system, it is unrealistic to refactor them for improved testability. It would simply not be cost effective. Hence, they must be tested as they are.

Yoo and Harman [1] recently conducted a survey on regression testing minimization, selection and prioritization, constituting nearly 200 papers. It encompasses the main research results around regression testing, addressing the problems of identifying obsolete, reusable and re-testable test cases (selection), eliminate redundant test cases (minimization) and order test cases to maximize early fault detection (prioritization). The survey shows that the majority of the works focuses on white-box testing strategies, concerning relatively small stand-alone programs written in C or Java, or for spreadsheets, GUIs and web applications. The techniques surveyed presuppose an already existing, effective test suite on which to select, minimize and prioritize test cases for the regression test. Before addressing these issues, we needed to take one step back to figure out how we should collect a test baseline, and how to perform regression testing.

Chays et al. [2] noted the lack of uniform methods and testing tools for verifying the correct behavior of database applications, despite their crucial role in the operation of nearly all modern organizations. Most literature in the field was aimed at assessing performance of database management systems rather than testing the database application system for functional correctness, let alone regression testing. The authors proposed a framework for functional testing of database applications called AGENDA [3-5]. However, the framework was not intended for regression testing and we found some of the ideas hard to scale, which had only been evaluated for smaller examples.

The most relevant work we found targeting regression testing for database applications was the SIKOSA project [6-8]. The authors proposed a capture-and-replay tool for carrying out black-box regression testing of database applications. This aligned well with our objectives regarding database regression testing, namely a capture-and-replay approach, similar to what has been more commonly used for GUI testing, to automatically identify differences between the results of two identical test runs (referred to as *deviations*). Because it is hard to build a precise test oracle for database applications with very complex queries, a more practical strategy is to capture a set of test case executions of the system under test, under the assumption that it currently works correctly (the *baseline*), and then use the replay run after modifications (the *delta*) to identify deviations and thus potential regression faults. Note that because such deviations only indicate *potential* faults, as they may also be due to valid changes, a technique is also needed to identify *actual* faults in a cost-effective manner.

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

We also needed a specification-based, black-box testing technique to help specify test input data (test cases) with adequate coverage, based on an analysis of the input domain for a given batch job. There are many suitable tools for this purpose, but we found that the classification tree modeling technique and the supporting tool CTE-XL [9], which is built on the well-known category-partition approach [10], was both easy to use and scaled up to the kinds of input domains under consideration (e.g., more than 100 categories or classifications in one model).

We also investigated Oracles Real Application Testing (RAT) [11], but found that it was mainly targeted towards performance testing and not easily adaptable for functional testing.

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

IV. DART

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

The basic principle of the tool is straightforward: Execute the system under test twice on the exact same input data and initial database state, once with the original version of the system (baseline), and once with the changed version of the system (delta). Compute the difference in database state between the two runs. A difference is either due to a valid change, or a regression fault.

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

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

Figure 1 shows the main steps in the testing process with DART. In the following sections, these steps will be described in detail.

A. *Running example*

Throughout the description of DART, a running example will be used to demonstrate the various steps of the test process. The example is intentionally kept very simple to fit size constraints. The system under test used as example is the program P shown in Figure 3. We use a Java-like syntax augmented with directly executable SQL statements in order to make it easier to understand for readers not acquainted with PL/SQL. It is a program that contains features for maintaining customer orders, more specifically adding and deleting items from a customer order. As an example execution of the program, one item is added to a customer order, while an item is removed from another customer order in the main method.

The relational entity model of the example program is shown in Figure 2, along with the initial state of the database prior to test execution. It consists of three entities containing information about customers and their orders. A customer can have zero to many orders with zero to many items.

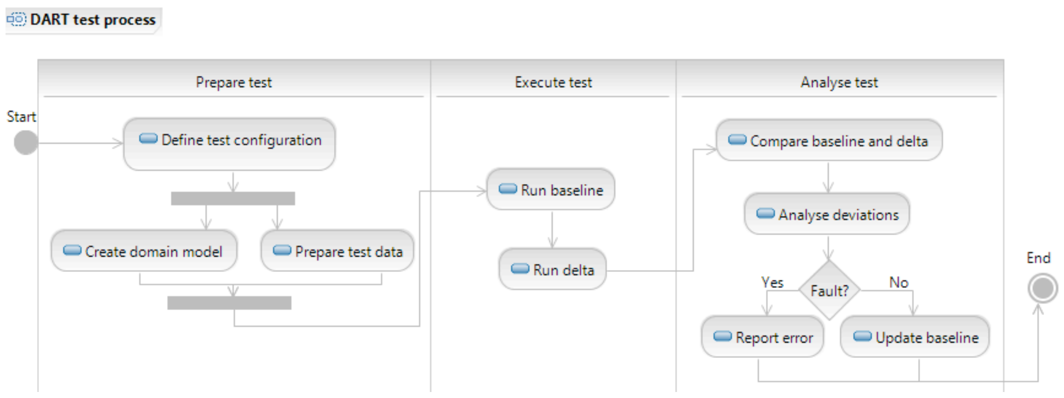

Figure 1: A UML Activity Diagram of the DART regression test process.

Program P

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

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

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

Figure 3: The example program P.

B. Test configuration

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

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

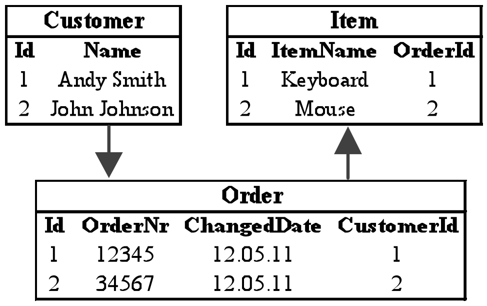

Figure 2: The relational entity model and initial state for program P.

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

TABLE I. TEST CONFIGURATION FOR PROGRAM P.

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

C. Domain modeling

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

In order to prioritize the test cases in the test suite, a model of the domain under test is made using the tool CTE-XL. The model is a classification tree (defining equivalence classes), which is used to generate domain partitions (also called test case specifications or abstract test cases) according to a coverage criterion of your choice, for example pairwise coverage of the equivalence classes. A domain model for the example program P can look like the one shown in Figure 5. The root node *Program P*, the classifications *Number of orders for customer, Item added* and *Item deleted*, and the classes (*0*, *1*, *>1*) and (*Y*, *N*) constitute the classification tree model, whereas the bottom six lines each represent *partitions*. In this case the pairwise

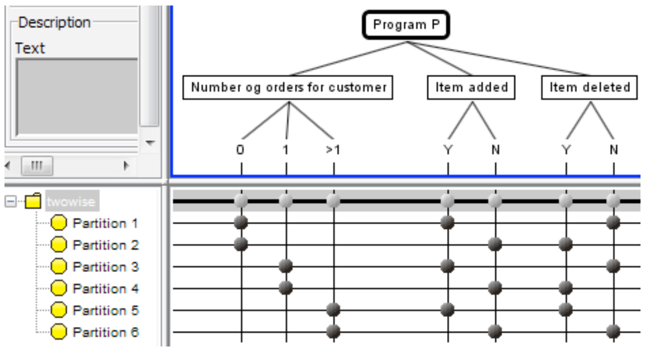

Figure 5: A classification tree model for program P.

coverage criterion was used to generate the partitions, which ensures that each pair of classes are represented in at least one partition. The test model emphasize the following aspects regarding the program P:

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

Given a test suite and a domain model of the system, DART provides the capability of matching the data in the test suite with the partitions in the domain model. In the initial state of program P shown in Figure 2, there are two test cases, namely customer *Andy Smith* and *John Johnson*. DART will match the test case Andy Smith with Partition 3 as he has one order in which an item will be added, and the test case John Johnson with Partition 4 as he has one order in which an item will be deleted. When the test cases have been matched to partitions, DART prioritizes the test cases as follows: Among the partitions containing test cases, select a random partition, and a random test case within the partition. Next, select again a random partition among the remaining ones, and again a random test case within the partition. Continue until all partitions have been selected. Then start the process again and select test cases among the ones that have not been selected yet. Stop the process when all test cases have been selected. The resulting ordering of the test case selection determines the priority of the test cases. The rationale is to ensure that all partitions be covered as quickly as possible during test execution and that, for cases where there is a deviation, the inspection of such deviations are more likely to uncover dissimilar regression faults as quickly as possible. This strategy should be considered as a first step to be improved upon, as further described in the conclusion.

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

D. Test execution

During test execution DART will log all data manipulations related to the specific test configuration. The way data manipulations are recorded and logged is through dynamically generated database triggers on the tables specified in the test configuration. A trigger is procedural code that is automatically executed in response to certain events on a table or view in a database. Pseudo-code for generating the triggers is shown in Figure 4. As the algorithm shows, a trigger is generated for each table in the test configuration. Each of the generated table triggers is defined to insert a row into the DART log table for each data manipulation on the columns specified in the test configuration for the given table. Insert and delete operations are always done at the row level and DART will log values for all table columns in the test configuration when an insert or delete operation takes place. Update operations can be attribute specific, so DART will only log the table columns in the test configuration that is actually updated. The triggers are dynamically generated as a Data Definition Language (DDL) string, which is executed in the end to store the actual triggers in the database.

Thus, DART dynamically *instruments* the database of the system under test by generating test-configuration specific database triggers when the test is started. During test execution these triggers will fire on any insert, delete or update on the table columns in the test configuration and store the database operations into a DART log table. One data manipulation operation results into one row in the log

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

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

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

Figure 4: Algorithm for trigger generation in DART.

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

A test run is done once with the original version of the system (baseline) and once with the changed version of the system (delta), which is subject to regression faults. Before the delta test run the database is reset to the initial (baseline) state to ensure that both runs start out with the same database state. Various mechanisms are available to reset the database. We have used the *flashback to restore point* feature of Oracle in the particular case of SOFIE. This is done by creating a restore point in the database after the test configuration is defined and the test data is prepared, but before the execution of the baseline run starts. The restore point defines the state of the database at the time it is created and will ensure consistency between the test runs.

In our example, a test run on program P, with the test configuration from Table I and the initial state from Figure 2, would result in the DART log data shown in Table II. For the test case *Andy Smith*, one insert operation and one update operation is executed, as logged in row 1 and 2 of Table II, respectively. For the test case *John Johnson* one delete operation and one update operation is executed, as logged in row 3 and 4 of Table II, respectively.

TABLE II. EXAMPLE DART LOG TABLE AFTER THE BASELINE RUN.

Id	Test Run Id	Test Case ID	Table Name	Column Name	Old Value	New Value
1	1	Andy Smith	Item	Item Name	Inserted	USB Stick
2	1	Andy Smith	Order	Changed Date	12.05.11	14.05.11
3	1	John Johnson	Item	Item Name	Keyboard	Deleted
4	1	John Johnson	Order	Changed Date	12.05.11	14.05.11

It turns out that program P contains a fault. The `changedDate` of the order should be updated to today's date when an order is changed. Currently it is updated to tomorrow's date. The fault is corrected (underlined) and a new version of P, called P' is shown in Figure 6. For illustration purposes let us assume a regression fault in P': the update of the order in `removeItemFromOrder` method is completely removed, rather than fixed (line struck through). After resetting the database into the same initial state as before the first test run, the test is executed again on the changed program version P'.

Program P'

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

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

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

Figure 6: The example program P', which is a modified version of program P.

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

TABLE III. EXAMPLE DART LOG TABLE AFTER BOTH TEST RUNS ARE EXECUTED

Id	Test Run Id	Test Case	Table Name	Column Name	Old Value	New Value
1	1	Andy Smith	Item	Item Name	Inserted	USB Stick
2	1	Andy Smith	Order	Changed Date	12.05.11	14.05.11
3	1	John Johnson	Item	Item Name	Keyboard	Deleted
4	1	John Johnson	Order	Changed Date	12.05.11	14.05.11
5	2	Andy Smith	Item	Item Name	Inserted	USB Stick
6	2	Andy Smith	Order	Changed Date	12.05.11	13.05.11
7	2	John Johnson	Item	Item Name	Keyboard	Deleted

E. *Test analysis*

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

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

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

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

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

TABLE IV. THE DEVIATIONS BETWEEN THE TEST RUNS FOR P AND P'.

Id	Test Case	Table Name	Column Name	Old Value	New Value	Test Run
1	Andy Smith	Order	Changed Date	12.05.11	14.05.11	Baseline
2	Andy Smith	Order	Changed Date	12.05.11	13.05.11	Delta
3	John Johnson	Order	Changed Date	12.05.11	14.05.11	Baseline

V. PRACTICAL EXPERIENCES

A. *Pilot evaluation*

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

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

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

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

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

TABLE VII SUMMARY OF TEST RUNS IN THE PILOT EVALUATION

Test	# Test cases	# Deviations	# Correct deviations	# Faulty deviations	# Distinct faults	# Previous faults found	# New faults found	Inspection time
1	711	33	19	14	7	5	2	7 hours
2	3144	182	136	46	11	7	4	35 hours
3	5670	522	386	136	15	6	9	105 hours
				Total	**17**	**8**	**9**	

For the purpose of the evaluation, we analyzed all deviations in the three test runs to ensure that we found as many defects as possible. However, this required a considerable amount of manual effort, as shown in Table VII (Inspection time); on average we used about 12 minutes per deviation. This suggests that, in order to use DART for large-scale regression testing in a system release, we would need a way to prioritize test cases to increase the likelihood of early fault detection and reduce the number of redundant deviations to analyze. The same functional fault was present in several deviations, and ideally we would only like to inspect one deviation for each unique functional fault. Thus, a classification tree model of the input domain was developed and applied to prioritize test cases, as described in Section IV.C.

We applied the prioritization to the test cases in test suite 3 as it was the largest. Figure 7 shows the results of using the partition-based approach for prioritizing test cases to execute *n* test cases and analyze the resulting deviations in their given priority order for various values of *n*. The results are then compared to the average resulting from the random selection of test cases. To obtain the results in Figure 7, we repeated the prioritization procedure 100 times and averaged the percentage of faults detected (the Y-axis) for a given percentage of test cases in the test suite (the X-axis). Though the results are very clear just by looking at Figure 7, to check the statistical significance of the difference between the partition-based approach over the random approach, we conducted non-parametric Mann-Whitney U-Tests [12] to test the difference in fault detection for each test suite size value. We computed p-values for all sizes that were sampled and all of them were below $\alpha = 0.05$, showing that the two approaches are significantly different. More precisely the p-value was less than 0.0000002 from 1 to 90 percent of the test cases, and 0.01381 for 95 percent. We tested the entire set of sample data from the two approaches, which yielded a p-value of 0.00019.

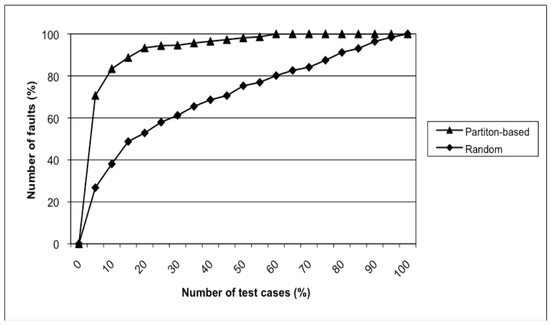

Figure 7: A comparison of partition-based-, and random test case selection.

In practice this means that for example by only executing eight percent of the test cases and analyze the resulting deviations, the test engineer would on average find more than 80 percent of the faults. This corresponds to executing approximately 450 test cases, which on average resulted in 80 deviations uncovering 12 out of 15 faults that could be uncovered by the test suite. In terms of effort that is 16 hours of inspection time for revealing 12 out of 15 faults. In comparison, with a random selection strategy, we would on average have found less than 35 percent of the faults for a similarly sized test suite. We consider this to be a substantial, practically important cost saving.

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

B. Test coverage and synthetic test data

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

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

The lack of model coverage achieved with live test data along with their associated practical concerns motivated the generation of synthetic test data. To drive the generation of synthetic test data, we use the same domain model as we use for partition-based test case prioritization. Adapter code is written to map the abstract values of the leaf classes in the classification tree model to actual parameter values of the real test cases for the system under test. Using the adapter code, test cases can be automatically generated according to the model. This makes it easy to generate different test suites, providing different levels of model coverage, e.g., two-wise, three-wise, or all combinations. The usage of synthetically generated test data with DART is still in its initial phase. A few system faults were identified while developing the adapter code, as rare system scenarios got executed. We are confident that the generation of synthetic test data will allow us to increase test coverage and make testing more efficient and predictable when applied in DART.

C. *Deployment into project setting*

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

The figures in Table V are meant to give a rough picture of the impact of DART during its initial lifetime in the SOFIE project. Unfortunately, we do not have exact information about the effort spent for uncovering the faults by the different testing approaches, as we faced organizational challenges in the project while trying to get the time reported at a satisfactory level for evaluation. However, the faults uncovered by regular testing are typically the result of weeks of testing, while the faults uncovered by DART result from days of testing. It is also worth stressing that we had no regression test environment in place in the first six releases shown in Table V. Consequently, DART was not used during the test period, but rather as a final verification of the releases after the acceptance test was finished and the release was ready to ship. Therefore, the figures provided in Table V should not be used to strictly compare the fault detection capabilities of DART with those of the regular testing routines, as DART could only detect the leftover faults in the first six releases. Table V show that DART has helped uncover more than a third of the defects found during regression testing (22 out of 59), within the batches of the core functional domain. Put in other words DART has helped identify approximately 60% more regression faults than what would have been detected without it. We consider this to be of substantial impact, especially since DART was only used as a "last check" in the first six releases. Such results combined with the savings discussed in Section V.A, make us confident that the test team can now rely on DART for regression testing of the batch jobs in SOFIE, while reassigning some of their resources on other types of testing. For example, faults in the graphical user interfaces, documents and reports within the same functional domain were discovered by regular testing routines, but would not have been found by DART. The same applies to the extensive testing required to verify the correctness of new functionality. An example of the latter is release six in Table V, where substantial new functionality was introduced, and thoroughly tested, revealing several faults in the regular testing routines.

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

Six of the defects were actually discovered by DART. One was not found as we ran the test on a limited scope in the beginning, before broadening our horizon the whole batch process of the functional domain in the later releases. Two of the faults were performance-related issues only present in the production environment (due to different settings). Besides the two currently unknown defects, that leaves us with three defects that should have been detected, but were not due to insufficient partition coverage. We hope to address this issue in the future by synthetically generating test data, as discussed in the previous section.

For the sake of the evaluation we also investigated the criticality distribution of the defects reported from manual

TABLE V. DEFECTS DETECTED IN THE PAST EIGHT RELEASES OF SOFIE

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

TABLE VI. REASONS WHY DEFECTS REPORTED FROM PRODUCTION WERE NOT FOUND BY DART.

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

testing, DART, and production. No conclusion could be drawn regarding the relationship between the criticality of defects and how they were detected.

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

VI. Conclusion and Future Work

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

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

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

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

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

Acknowledgment

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

References

[1]. Yoo, S. and M. Harman, *Regression testing minimisation, selection and prioritisation: A survey.* Journal of Software Testing, Verification and Reliability, 2011, 2011. **To appear.**

[2]. Chays, D., et al., *A Framework for Testing Database Applications.* ACM SIGSOFT Software Engineering Notes, 2000. **25**(5): p. 10.

[3]. Chays, D. and Y. Deng, *Demonstration of AGENDA tool set for testing relational database applications*, in *International Conference on Software Engineering*. 2003, IEEE Computer Society: Portland, Oregon. p. 802-803.

[4]. Chays, D., et al., *An AGENDA for testing relational database applications.* Software Testing, Verification & Reliability, 2004. **14**(1): p. 28.

[5]. Deng, Y., P. Frankl, and D. Chays, *Testing database transactions with AGENDA*, in *International Conference of Software Engineering*. 2005, ACM: Association for Computing Machinery: St. Louis, MO, USA. p. 78-87.

[6]. Haftmann, F., D. Kossmann, and A. Kreutz. *Efficient regression tests for database applications*. in *The Conference on Innovative Data Systems Research (CIDR)*. 2005. Asilomar Conference Grounds, Monterey Peninsula in Pacific Grove, CA, USA.

[7]. Binning, C., D. Kossmann, and E. Lo. *Testing database applications*. in *International Conference on Management of Data*. 2006. Chicago, IL, USA: Association for Computing Machinery (ACM).

[8]. Haftmann, F., D. Kossmann, and E. Lo, *A framework for efficient regression tests on database applications.* The VLDB Journal — The International Journal on Very Large Data Bases, 2007. **16**(1): p. 145-164.

|9]. *CTE XL and CTE XL Professional - Overview*. Available from: http://www.berner-mattner.com/en/berner-mattner-home/products/cte-xl/.

|10]. Ostrand, T.J. and M.J. Balcer, *The category-partition method for specifying and generating fuctional tests.* Magazine Communications of the ACM, 1988. **31**(6).

|11]. *Oracle Real Appliaction Testing*. Available from: http://www.oracle.com/us/products/database/options/real-application-testing/index.html.

[12]. Arcuri, A. and L. Briand. *A practical guide for using statistical tests to assess randomized algorithms in software engineering.* in *International conference on Software engineering.* 2011.