Modelling and Verifying Combinatorial Interactions to Test Data Intensive Systems

Experience with Optimal Archiving at the Norwegian Customs and Excise Directorate

Sagar Sen, Dusica Marijan, Carlo Ieva, Astrid Grime, and Atle Sander

***Abstract*—Testing data-intensive systems is paramount to increase our reliance on information processed in e-governance, scientific/medical research, and social networks. Data accrued in these systems often go through several manual and computational steps involving human inputs in interactive media and complex batch applications that aim to ensure high quality of data in terms of validity, correctness, and adherence to business rules. Testing these systems involves verifying data in test databases, copied from different steps of *live production streams*, for adherence to business rules. We simplify the process by modelling and automatically generating relevant test cases as data interactions satisfying the combinatorial interaction coverage criteria. We verify these test cases using our human-in-the-loop tool, DEPICT. DEPICT, with expert assistance, generates complex SQL queries for test cases and produces a visual report of test case satisfaction. We apply the approach to simplify and optimize a periodic archiving operation within the testing environment of the Custom directorate's TVINN system.**

***Index Terms*—Classification Tree Modelling, Data Interactions, Combinatorial Interaction Testing, Human-in-the-loop, Relational Databases, Testing, E-governance**

ACRONYMS AND ABBREVIATIONS

SQL	Structured Query Language
TVINN	TollVesenets INformasjonssystem med Nringslivet
DNCE	Directorate of Norwegian Customs and Excise
HTML	HyperText Markup Language
JDBC	Java Database Connectivity
CUSDEC	Customs Declaration
CUSRES	Customs Response
CTM	Classification Tree Model
KID	Customer Identification Number
SSB	Statistical Bureau of Norway

NOTATION

Ct	Classification tree
A, B, C	Classifications
a, b, c	Classes
t_c	Pairwise-covering test cases
l	Dependency rule

S. Sen, D. Marijan, and C. Ieva are with Certus V&V Centre, Simula Research Laboratory, Norway, e-mail: sagar@simula.no
A. Grime and A. Sander are with the The Norwegian Customs and Toll, Norway.

I. INTRODUCTION

Data-intensive software systems are increasingly prominent in driving global processes such as e-governance, data curation for scientific and medical research, and social networking. Large amounts of data is collected, processed, and stored by these systems in *databases*. For example, the Directorate of the Norwegian Customs and Excise (DNCE) uses the TVINN [1] system to process about 30,000 customs declarations a day coming in from both individuals and enterprises. The live transaction stream of declarations is processed for conformance to well-formedness rules, customs laws and regulations by complex batch applications. This scenario is prevalent in many data-intensive software systems dealing with *transaction data* which comprises semi-structured/structured data in medium/high volume.

Verifying databases is an integral part of the testing process in data-intensive systems. From our industry experience at DNCE we observe that the common practices to verify databases include (1) domain users observing data in a customized user interface and verifying it against a checklist or matrix of requirements in a spreadsheet. The semi-systematic approach of using spreadsheets as a structured checklist for properties to be found in test artifacts (such as test databases) has been shown to be effective in industrial practice [34]. However, documenting testing intentions in a spreadsheet does not associate clear computable meaning to the tests making them ambiguous and a challenge to maintain; (2) skilled but few database experts create, often complex, structured query language (SQL) queries to verify data in databases. This approach is often tedious, error-prone [32], database technology specific, fails to capture high-level testing intentions much needed for documentation, and consequently hard to maintain [22]. Moreover, studies assessing human factors have shown that it is not easy to create such queries, especially when multiple table joins are involved [48] [32] [25]. These observations led us to believe that specifying requirements to verfiy databases *must be raised to a higher level of abstraction*. **Previous Work:** Our first objective was to introduce *model thinking* to raise the level of abstraction in representing realistic scenarios in test databases. In recent work, [40] we proposed modelling and verifying realistic scenarios as *data interactions* to improve testing of data-intensive systems. The intuition stems from the observable interaction between data elements in testing intentions such as, "If **whisky** is

[1] http://toll.no/

imported then it needs to be **manually** processed by a customs officer" or " If **net weight** is greater than 1/3 **gross weight** then the item must have been **manually** inspected". The text in bold represents information stored in fields belonging to different tables in a database. We model data interactions [37] using the *classification tree modelling* formalism [15] and tool TESTONA [2][23]. We associate semantics or computable meaning to these models using a human-in-the-loop tool DEPICT. DEPICT first queries database meta-data to extract a graph of the database schema. DEPICT provides an *interactive interface* to help a tester specify a *connected subgraph* between the different nodes (representing tables) based on relationships such as: (a) referential integrity (primary key - foreign key relationships), and (b) surrogate and self-referential relationships based on type matching between one or more fields between two tables. After a valid connected subgraph is created, DEPICT automatically generates a query for each test case. Each query results in an *inner join* between the tables of the connected subgraph with *where* clauses to equate classes (from the test case) to each field. The resulting *interaction table* represents test cases that are covered by the test database, and DEPICT counts the number of occurrences and produces an HTML report of the coverage. DEPICT is implemented as a standalone Eclipse [35] rich-client platform application and uses a non-vendor specific subset of JDBC [49] for database connectivity rendering the tool database technology agnostic. We evaluated several models and the tool DEPICT in [40] with successful replication of real test scenarios as models and their verification with DEPICT.

Challenge to Increase Automation: Modelling test cases was still a manual task as reported in [40]. Test managers at DNCE complained about the *repetitiveness and tediousness* of creating test groups and test cases in a classification tree model. Can we specify a modelling domain such that all test cases can be automatically generated and verified against a test database? This is the question we address.

Contributions: We present the idea of *Combinatorial Interaction Testing for Data Intensive Systems* in this paper. Testing with interactions and combinations is not a new idea in software testing [13] [52]. However, the focus on data itself and the possible interactions between datum in a database is, to the best of our knowledge, a less explored direction. We outline the contributions of the paper (extending the work previously presented at ISSRE 2014 [40]) as follows:

1) **Modelling Domain of Data Interactions based on Dependency Rules:** We consider a database as a software artifact under test. Using the classification tree modelling formalism with boolean dependency rules we model the constrained domain of data interactions (described in Section III). The advantage of modelling with dependency rules is the possibility to automatically generate meaningful test cases or data interactions which are correct by construction. This is a conceptual contribution (not a technological one) towards representing a modelling domain of test cases for databases using classification trees.

[2]http://www.testona.net/en/

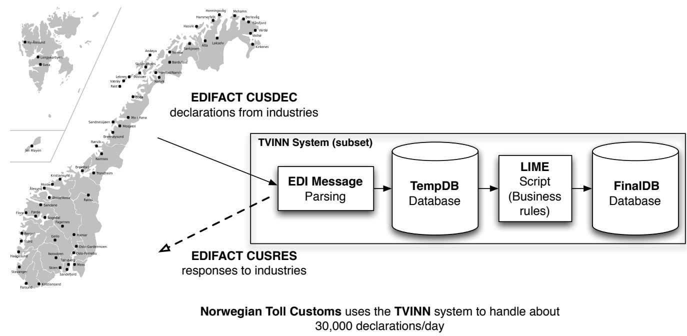

Fig. 1. Overview of the TVINN system at the Directorate of Norwegian Customs and Excise

2) **Automatic Test Generation based on Combinatorial Interaction Coverage:** We automatically generate test cases for a model satisfying both dependency rules and a combinatorial interaction coverage criteria. The advantage of this method is that a tester can selectively (by activating a dependency rule) generate test cases with pairwise coverage without having to manually specify them. The contribution aims to increase automation in the testing process of a data-intensive system.
3) **Application to Optimal Archiving Operation at DNCE:** We apply our approach to optimize the archiving operation (described in Section V) within DNCE's test environment. Archiving can only take place when customs declarations have been through a precise sequence of operations reflected by several flags in TVINN's test database. We automatically generate test cases to represent the different scenarios and use DEPICT to monitor the database in the testing environment. For instance, test cases ensure that (a) a declaration has received a payment (b) content from declarations have been sent to the statistical bureau of Norway, to name a few. Lessons learnt (in Section VI) from this experience demonstrate the advantages of modelling and combinatorial testing to simplify and increase the efficiency of the testing process at DNCE.

The paper is organized as follows. Background work (from [40]) is spread out in large parts of the paper that is necessary for a comprehensive understanding of our experience in combinatorial interaction testing at DNCE. In Section II, we describe both a simplified running case study and industrial case studies from DNCE. In Section III, we review modelling data interactions with classification tree models. Section IV presents an overview of the tool, DEPICT to verify test databases using classification tree models as input. We apply our approach to a periodic archiving scenario at DNCE in Section V. Lessons learnt from this experience are discussed in Section VI. Section VII discusses the related work and places our contribution in the body of knowledge. We conclude in Section VIII.

II. CASE STUDY

TVINN is a Customs information system for business and trade developed by and for the Norwegian Customs as shown

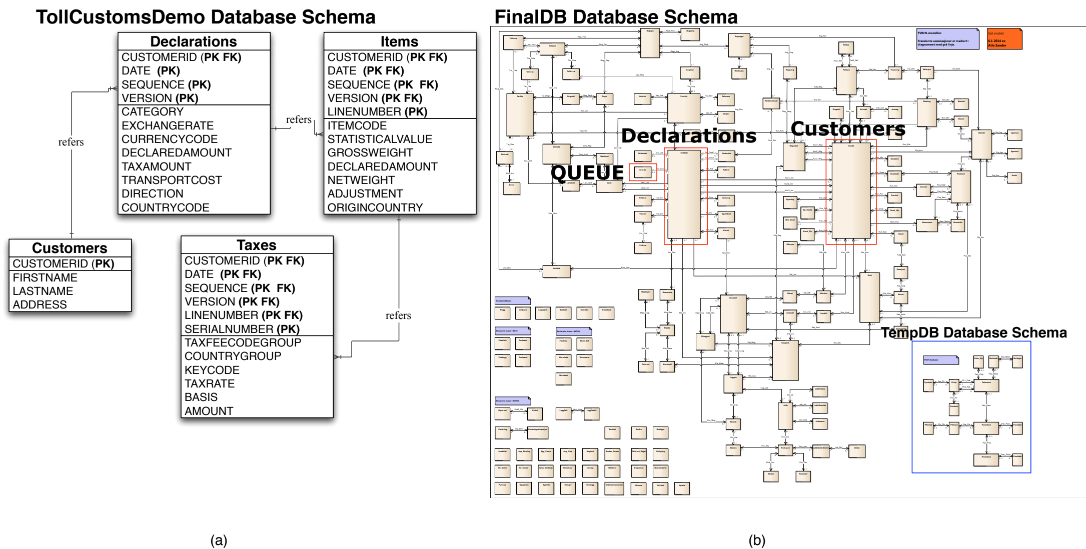

Fig. 2. (a) Simplified Database Schema TOLLCUSTOMSDEMO at Norwegian Toll Customs in Crow-Foot Notation (b) Bird's Eye View of FINALDB and TEMPDB Database Schemas (reproduced from [40])

in Figure 1. All customs declarations regarding import and export to and from Norway are processed by this system and 98% of them are received electronically by use of the EDIFACT[3] standard. Incoming declarations are received as CUSDEC (Customs Declaration) messages and outgoing responses are sent in CUSRES (Customs Response) messages. During weekdays, the number of incoming CUSDEC messages is approximately 25-30,000. Each incoming message is subject to input control, with different checks, such as:

- Conformity to the protocol used, EDIfact;
- Optional, mandatory and conditional parameters;
- Correct values for parameters specified;
- Static and dynamic filtering based on message data;
- Business rules.

Some of these checks will trigger the system to accept the declaration, but initiate manual control by a customs officer. Other checks might cause the message to be rejected by the system. If no checks trigger any specific action but approval, the message is processed automatically by the system. Independent of the outcome from above, a response message with the result is automatically returned. The response will always include one or several unique numerical codes (can be fault code) identifying the result. Parsed EDIFACT messages from live transactions are stored in the TEMPDB database. While, customs declarations validated by the principal batch application LIME (see Figure 1) are stored in the complex and highly structured FINALDB database. The *principal challenge* is to verify that the TEMPDB and FINALDB databases have correctly executed the above mentioned checks. The challenge is evergreen since checks in TVINN evolve on a regular basis (approximately, every six months), depending on new governmental policies, sanctions, and change in political parties. TVINN is also affected with time-bounded rules created by customs officers. These rules exist for a short period of time (often 3 months). For instance, a customs officer could decide to thoroughly check items and consult seniors for imports with no prior tax laws for a fixed period of time. These kinds of rules are called *filter control* and can be disabled/deactivated after a fixed time limit. These rules can change on an everyday basis, without anticipation, making TVINN a highly dynamic system.

Testing TVINN has been achieved by a small testing staff who manually executes the tests and verifies the results. However, the current practice of using such a test database presents three crucial problems:

No Coverage Guarantee: Live declarations are expected to cover a realistic subset of the database's domain (set of all possible combination of values in fields and tables of a database). However, there is no way to guarantee this coverage in an effective manner.

Very Large Set of Test Records: Accumulating information from live transactions can easily give rise to an ever-growing set of data records in a test database. Many of these records share similarities and hence are redundant for the purpose of testing. Cost-effective testing will require a verification of a test database for the number of occurrences and consequently selecting only a minimal set of records in a test database. A minimal set will also have modest time and space requirements for testing efforts.

Constantly Changing Rules: Test databases have a lifetime and need to be discarded. For instance, this may be needed due to new governmental policies such as increase in taxes on imported cheese, or when sanctions are imposed on certain countries. Legacy test databases may not be used anymore to test the evolved system. Therefore, they need to be constantly verified for testing adequacy.

We address the above problems by combinatorial interaction testing using our DEPICT in large test databases from TVINN. In Section II-A, we describe a simplified version of the TVINN's test database that we use as a running example throughout the paper. In Section II-B, we describe the complex

[3] http://www.unece.org/trade/untdid/welcome.html

industrial TEMPDB and FINALDB databases that we evaluate in Section V to demonstrate the industrial relevance and scalability of our approach.

A. Simplified Test Database

As a simplified running example, we present a schema developed along with our industry partner, the Norwegian Customs and Excise, in Figure 2(a). The database schema for TOLLCUSTOMSDEMO consists of *four tables* and is created on a MySQL server. We describe the tables and some of the fields in them. The CUSTOMERS table is used to store records of customers. A customer is identified by a CustomerID which is a primary key (indicated PK). A customer can make one or more declarations. These declarations are stored in the DECLARATIONS table that *refer to* a customer using a foreign key (indicated FK). A declaration can have one or more items that are stored in the ITEMS table. Every item has an item code and a statistical value of its cost. There can be different types of taxes on an item which is stored in the TAXES table. The most common form of tax is the value added tax or VAT. Taxation rules are often expressed on the country group, tax fee code group (from the TAXES table) of import and the item code (from the ITEMS table). The 10,000 items codes, 88 country groups, and 934 tax fee codes can potentially give rise to 12.9 trillion 3-wise possible taxation laws. However, only 195,000 taxation laws are used in practice.
Size: The test database used as a running example contains about **363 customers, 8172 declarations, 60591 items, and 63515 tax records**.

B. Industrial Test Database

The schema of the industrial test database is confidential. However, we present a bird's eye view of the relational database schema in Figure 2(b) for the TEMPDB and FINALDB databases.The database TEMPDB contains raw data from EDIFACT messages. It has **18 tables, 188 fields, of which 13 are relationships** between primary and foreign keys.

The principal database FINALDB consists of **132 tables, 1335 fields, of which 157 are relationships** between primary and foreign keys. It is the far more complex industrial version of the simplified schema shown in Figure 2(a). The complex database reflects more than just one possible path between two tables complicating the task of selecting a path relevant to specific scenario. For instance, the tables DECLARATIONS and CUSTOMERS have three PK-FK associations as shown in Figure 2(b): (a) A customer is the legal owner of the declaration, (b) A customer is the declarant or importer, and (c) A customer is the payer for taxes on the declaration. Establishing an interaction between a field in CUSTOMERS and a field in DECLARATIONS will require an informed selection between the three possible paths. Manually creating an entry into the FINALDB database requires a specialist who can navigate his/her way through a large number of dialogue windows. This complex operation explains that using an existing test database and knowing exactly what missing information to insert into a test database can greatly reduce manual effort. Completed customs declaration in DECLARATIONS are put into a queue for archival to a write-once-read-many-times (WORM) backup drive to keep query performance optimal in FINALDB. The state of a declaration before it is ready for archival is available in the table QUEUE.
Size: The test databases FINALDB and TEMPDB contain about **2.5 million customers and 190,000 declarations**. In addition, it contains **93,000 legal documents** sometimes used to justify taxes.

III. MODELLING DATA INTERACTIONS

Modelling data interactions was introduced by the authors in [37]. Data interactions are modelled on a relational database schema. We briefly describe a database schema in the following Section III-1. We use the classification tree modelling formalism to model data interactions as presented in Section III-2. The content of Sections III-1, III-2 has been reproduced from our previous work [40].

1) Database Schema: Databases are typically modelled using a data model such as a *database schema*. It specifies the input domain of a database in an information system. We briefly describe the well-known concept of database schema. More information on them can be found in a standard database textbooks such as [10]. A database schema typically contains one or more *tables*. A table contains *fields* with a *domain* for each field. Typical examples for field types/domains are integer, float, double, string, and date. The value of each field must be in its domain, hence maintaining *domain integrity* in a database. A table contains zero or more *records*, which is a set of values for all its fields within their domain. A table may also contain one or more fields that are referred to as *primary keys*, which identify each record. In addition, each table may refer to primary keys of other tables via *foreign keys*. The value of foreign keys must match the value of a primary key in another table. This is known as a *referential integrity* constraint. We refer to the combined concepts of *referential integrity* and *domain integrity* as *data integrity*. Records in a database must satisfy data integrity as specified by its database schema. Databases can be queried using SQL queries. We use queries to create inner-joins, views and to count number of records. An example of a database schema from DNCE is shown in Figure 2(b). The schema has four tables with three referential integrity constraints associating them, hence creating the possibility of interactions between these tables.

2) Classification Tree Model: Classification tree models are typically used to model the input domain or a subset of it for a software system. They contain the concepts of compositions, classifications, and classes for each classification to structure the input domain. We use the tool TESTONA [23] (previously known as CTE-XL) to model classification tree models (CTM). CTMs have been used to model complex input domains such as the domain of the Norwegian Tax Department [33]. We use CTMs to graphically model test cases for databases. Our use of the CTM is specialized to the need of specifying data interactions on a database. Hence, we first assume that a modeller has access to the relational database schema of the information system for which data interaction test cases need to be specified. We use the example in Figure

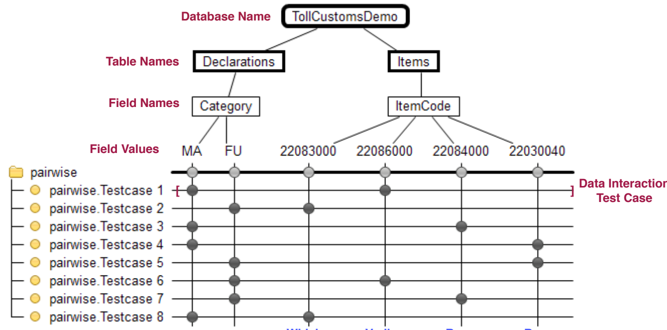

Fig. 3. Classification Tree Model for Alcohol Imports (reproduced from [40])

3 to describe the syntax of the specialized model as follows:

Root Composition for Database: It is an identifier for a database on a server. Software that analyzes the classification tree model can identify a concrete database using this identifier. In Figure 3, this is represented by the composition TollCustomsDemo.

Compositions for Tables: The root composition can contain several compositions representing identifiers for tables. In Figure 3, this is represented by the compositions Declarations and Items from the schema shown in Figure 2(b). All or only a subset of tables maybe be specified depending on the use of the model.

Classifications for Fields: Fields in tables are classifications in the third level of the classification tree. For instance, in Figure 3, we use the fields Category and ItemCode. All or only a subset of fields for a table might be specified in the model.

Classes for Fields: A field can have one or more concrete values or domains of values. We represent these possibilities as classes. For instance, in Figure 3, the classes MA and FO are associated with the classification for the field Category. At a rudimentary level, one may specify individual values for fields as a class, but this often leads to an explosion in the size of the model. Therefore, we provide special classes for domains of a field as shown in Figure 4. The most common special classes include the set operator IN, NOT IN, the pattern matching operator such as LK, NOT LK.

Interactions as Test Cases in Groups: Interactions between field values across different tables are represented as test cases in a classification tree model. For instance, in Figure 3, pairwise.TestCase1 represents the interaction {MA,22086000}. This is the interaction between a declaration category for manual processing of an import and for itemcode 2208600 for Vodka. Interactions in TESTONA can either be generated automatically such that all pairs or three-wise interactions between two or three classes are covered. In Figure 3, we present all pairwise interactions between a set of database field values. Testers can also manually specify them. Test cases or interactions can be divided into groups to represent test cases for different aspects of the information system. A *good practice* is to create *several small* classification tree models with test groups containing a small number of test cases with very specialized testing intent. This also renders the model-based documentation more comprehensible.

Syntax	Description	Example
*	Don't care condition. It means that the test cases which have this value selected won't take into account the class (column) whose value is '*'	
LK(<pattern>)	Filter condition by the means of the wildcards '%' and '_'. The '%' wildcard is a substitute for zero or more characters while '_' is a substitute for single characters.	LK(N%): all values starting with 'N'. LK(%N_): all values starting with any sequence of characters and ending with a 'N' followed by a single character.
NOT LK (<pattern>)	Negates the LK(<pattern>)	NOT LK(N%) NOT LK(%N_)
RNG(<number1>, <number2>)	A range over numeric values: number1 <= X <= number2	RNG(12.5, 15): 12.5<=field<=15. RNG(12.5,): X >= 12.5 RNG(,15): X <= 15
IN(<a,b,...,n>)	Express a set of predefined values: $X \in \{a, b, ..., n\}$	IN(8000,8015,8030)
NOT IN(<a,b,...,n>)	Negates the IN(<a,b,...,n>) function	NOT IN(8000,8015,8030)
ISNULL	Check whether the field is not NULL (not initialized)	
NOT ISNULL	Check if the field value is not NULL (initialized)	
ISBLANK	In string fields check whether the value is equal to the empty string ''	
NOT ISBLANK	In string fields check whether the value is NOT equal to the empty string ''	
ISEMPTY	Check, in string fields, whether the field value is NULL **or** blank	
NOT ISEMPTY	Check, in string fields, whether the field value is not NULL **and** not blank	
REC(<P_C, S_C>)	Filter rows over a recursive relationship. The first parameter defines the clause for the primary side of the relationship (the one used with further relationship) and the second defines the clause for the secondary side. A parameter can be any of the clauses listed above.	REC(CUSRES,CUSDEC) REC(,IN(EB,RE,SO)) REC(RNG(2,),) REC(LK(%IN),LK(%OUT))

Fig. 4. Special Clauses for Data Classes (reproduced from [40])

TESTONA allows specification of additional boolean constraints or dependency rules between classes to limit the number of interactions or test cases that can be specified by construction.

We may wish for a test case to exist or not in a test database. This is exactly what is verified by DEPICT, and the principal contribution of the paper. The method behind DEPICT is described in Section IV.

3) Generating Combinatorial Interactions: A test case for classification trees is defined by combining classes from different classifications. The length of the test case depends on the number of classifications of each class.

Let $C_t = (A[a_1, ...a_n], B[b_1, ..., b_m], C[c_1, ..., c_k])$ be a classification tree with three classifications A, B, and C, each classification with a number of classes. A **pairwise-covering** test cases tc is defined as a combination of classes taking every pair of classes from disjunctive classifications at least once, $tc = (a_i, b_j, c_k), i \in 1..n, j \in 1..m, k \in 1..k$. Similarly, a **threewise-covering** test case is defined as a combination of classes taking every triple of classes from disjunctive classifications.

Specifying dependencies: Dependency rules are used to specify constraints between the values of different classes. In test generation, dependencies restrict possible combinations of classes of a classification tree. For example, a dependency rule $l = [a_1, b_1]$ will require that the class b_1 is selected if the class a_1 is selected. Dependency rules can be defined for different logical dependency relations (AND, OR, NOT).

TESTONA provides an in-built mechanism to generate test cases that satisfy pairwise and threewise coverage with the possibility of selectively applying dependency rules. We use this in-built feature to generate a large number of data interactions without having to manually select classes in classifications of field values.

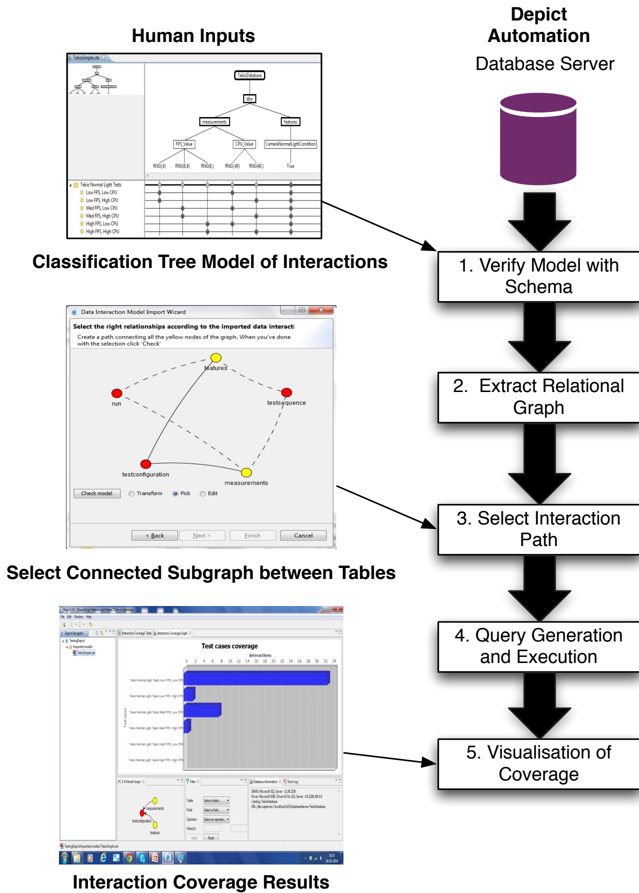

Fig. 5. Verifying Interaction Coverage (reproduced from [40])

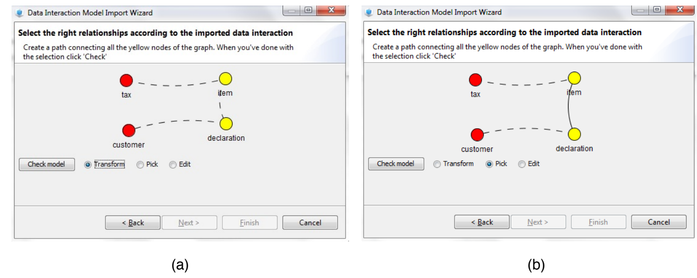

Fig. 6. (a) Relational Graph Extracted from Database Schema (b) Selection of Connected Subgraph

IV. MODEL VERIFICATION

Given a model of data interactions as input, our method computes the satisfaction or non-satisfaction of these interactions in a test database. An overview of the operational flow of our method is shown in Figure 5. The method has an automated part implemented in the Eclipse-based tool DEPICT and also requires human inputs to guide the process. Note that the approach shown in Figure 5 is about modelling and verification of data interaction, which is only a subset of the global testing process associated with TVINN. Figure 5 refers to background work already presented in [40]. The different steps of the method are described below:

Step 1: The input classification tree model (CTM) with test cases is first specified by a domain expert shown in Figure 3. We use the tool TESTONA (previously CTE-XL[23]) to create the model. The CTM is then imported into DEPICT. While importing a CTM, DEPICT requires additional parameters to connect to a database on a local/remote database server. DEPICT verifies that the CTM contains valid names for a database, tables, and field names. This is done by querying and comparing meta-information from the database schema on a database server. DEPICT's connection to a database is implemented using a non-vendor specific subset of JDBC [49], to facilitate connecting to different database technologies.

Step 2: DEPICT extracts a *relational graph* by querying the meta-data of the database schema using the JDBC class ResultSetMetaData [49]. For instance, in Figure 6(a) we show the graph extracted for the simplified schema shown in Figure 2(a). The yellow nodes in Figure 6(a) indicate the tables specified in the classification tree model of Figure 3. The other tables are shown as red nodes. The associations between the tables in the form of primary key and foreign key relationships are shown as dashed edges in the graph. The dashed edges highlight *potential arcs* between tables. One must manually select an edge to establish a relationship between nodes. DEPICT uses the JUNG Universal Graph Library[4] to display, perform automatic layout, and enable human interaction with the graph.

Step 3: The human domain expert defines a *connected subgraph* between interacting tables or nodes in the relational graph obtained in the previous step. This is illustrated by the edge created between yellow nodes for ITEMS and DECLARATIONS in Figure 6(b). The example in the figure represents the simplest possible interaction where two nodes can interact due to PK-FK relationships. If two nodes are disconnected, DEPICT allows the creation of a surrogate relationships (comprehensively illustrated in Scenario 2 of Section V), based on a type match between one or more fields of the two nodes/tables. For instance, we may create a relationship between the fields origin country in the ITEMS table and country code in the DECLARATIONS table for the schema in Figure 2(a). Both these fields have the same type which is an enumeration of 160 country codes. DEPICT also allows specification of self-referential relationships between fields of a unique node/table (also illustrated in Scenario 2 of Section V). A surrogate relationship can also be created between two tables even if they already have one or more PK-FK associations and are not completely disconnected. The advantages of creating surrogate relationships are:

- Creating totally brand new relationships which are not strictly functional to the data model of the system but having a valuable meaning for the testing perspective such as: better performance avoiding unnecessary complex paths through a shortcut across tables.
- Fill gaps into the original design of the data model. Sometimes the PK-FK relationships simply do not exist. In that case we are able to fill the gap creating a surrogate relationship.

The principal task of a human-expert is to select edges from

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

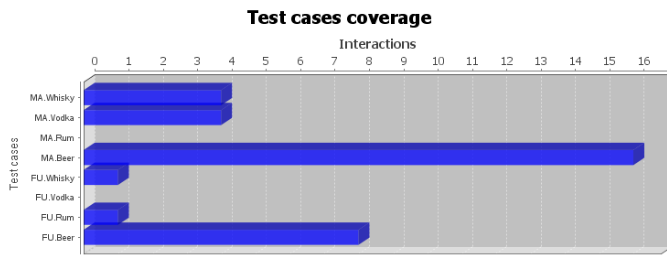

Fig. 7. Interaction Coverage for Alcohol Import Example (reproduced from [40])

potentially several possibilities, in order to create a connected subgraph that is meaningful. We present such an example in the industrial Scenario 1 in Section V.

```
SELECT declarations.customerid AS f_1,
declarations.date AS f_2,
declarations.sequence AS f_3,
declarations.version AS f_4,
items.customerid AS f_5, items.date AS f_6,
items.sequence AS f_7,
items.version AS f_8,
items.linenumber AS f_9
FROM declarations JOIN items ON
declarations.customerid=items.customerid
AND declarations.date=items.date
AND declarations.sequence=items.sequence
AND declarations.version=items.version
WHERE declarations.category='MA'
AND items.itemcode=22086000
```

Listing 1. Generated Interaction Table Example

Step 4: The connected subgraph between all interacting nodes is used by DEPICT to generate SQL queries, in order to create an *interaction table* for each test case. For instance, we generate 8 interaction tables for the model in Figure 3. The query to generate an interaction table for the test case pairwise.Testcase1 in Figure 3 is shown in Listing 1. The interaction table is stored with new field names $f_1, f_2, ..., f_n$ to avoid conflicts with tables that may have duplicate names. The table should contain all records where the field values in the CTM match the selected classes, which is declarations.category='MA' and items.itemcode=22086000. Additional queries are also generated to compute the frequency of the occurrence of a test case. These queries are executed by DEPICT on the test database to populate the interaction table, and compute the frequency of occurrence.

Step 5: We refer to test cases that are not found in the test database as "holes". This is indicated by a count of zero for number of records on an interaction table. If a test case is found in a test database then the count is non-zero. DEPICT, provides an HTML report to visualize the results of the queries executed in Step 4. The report provides a table with test case id (indicated by a yellow colored box, in case of "hole"), test case name, count, a human-readable expression of the test case, an SQL query for the test case, and the elapsed time to query the test database. DEPICT also generates a bar graph produced using JFreeChart[5] to display the count for each test case as shown in Figure 7 for the CTM in Figure 3. The report gives instant feedback to a tester about interactions that are missing in a test database and need to be created for complete coverage.

[5]http://www.jfree.org/jfreechart/

The tool DEPICT is implemented in pure Java as a standalone Eclipse Rich-client Platform application. We request the reader to contact the authors for instructions to download, install and use the tool for databases in their information systems. DEPICT currently supports MySQL, PostgreSQL, Sybase, Oracle, Microsoft SQL, and can easily be extended to other relational databases with the appropriate driver. DEPICT is a generic industry-strength tool, as demonstrated in Section V, and can handle databases with millions of records in a stable manner.

V. EXPERIMENT

We evaluate our approach to model-driven combinatorial interaction testing for the customs declaration archiving process at DNCE and discuss lessons learnt. This section presents a novel application of our method to monitor and verify test databases using the test cases satisfying combinatorial interaction coverage.

A. Optimal Archiving

The database TVINN, described in Section II-B, must be kept as small as possible for efficient query execution. The archiving batches perform a data cleanup by moving declarations into an archival database. However, a declaration must be verified through a number of steps before it is archived. In production, the flow of TVINN is complete when all batches are scheduled to process data from the customs declaration in the correct and natural order:

Step 1. Declarations are received, processed and responses are returned, invoices are generated (with customer identification numbers or KID numbers).

Step 2. Payment may require to wait for all online processing done by the customer officers to terminate. Customs officers define time-bounded masks to control specific types of customs declarations. In addition they may have to manually control incoming declarations.

Step 3. Invoices are sent to customers who will pay their bill (referencing this KID) and we will receive the money (OCR from the banks referencing this KID), match with the KID and then return the receipt if OK.

However, in the testing environment, a tester must regularly monitor *status of declarations* from production (during Step 2 above) to consequently test batch applications. Archival of a declaration depends on a combination of: (a) values in the field (FINALDB.DECLARATIONS.STATUS) of the table DECLARATIONS, and (b) values in several fields in the queuing state table FINALDB.QUEUE (shown in Figure 2 (b)). We discuss the different fields (names have been changed from original Norwegian for privacy reasons) and their possible values below:

FINALDB.DECLARATIONS.STATUS: A declaration is considered finished/completed and ready for archival when the status field has a value greater than 89 and less than 100 (97 not being used). For example, when a declaration has been finalized due to a newer version sent by a customer, it is identified by the value 94.

FinalDB.QUEUE.RECALC: A finalized declaration might be subject to *recalculation* due to many reasons. If the recalculation has started QUEUE.RECALC is set to V. If the recalculation is processed and finalized then QUEUE.RECALC is set to V again since it is an intermediate state indicator. When recalculation is finalized, the attribute will be updated in the table QUEUE.

FinalDB.QUEUE.RECEIPT: Declarations might need to be finalized for payment of fees, some can be archived immediately while some cannot be archived immediately until the payment is done. Its faster to delay archiving rather than extracting from the archive. The QUEUE.RECEIPT values U, Z implies not paid while NOT(U,Z) implies that fees are paid. RECEIPT is updated to U when customer pays. Status is Z when a receipt is being created while * means that receipt is not necessary or is 'do not care'.

FinalDB.QUEUE.EXTERNAL: Declarations must be sent to external entities such as the Statistical Bureau of Norway [6] (SSB) before archival. The value of EXTERNAL = J means ready for analysis/transfer, O is acceptance but not yet available for archiving, values I and X imply don't modify declaration, F is a fault, U means just arrived in the queue, or NULL which means no value for this attribute. A batch is run to dump all declarations into a file that is sent to SSB.

In the following section we describe how model-driven combinatorial interaction testing helps represent and test data interactions, that helped test managers at DNCE ensure correct and optimal archiving.

B. Approach Application

The first step in our approach is to model the domain of data interactions that need to be tested for archiving. In Figure 8, we present the classification tree model that represents the different possible values for the fields discussed in Section V-A and the test cases that cover all pairwise interactions. This model was created by a test manager at DNCE during the periodic archiving operations.

The test manager, instead of generating the set of all test cases (covering pairwise) using Testona, decided to divide the test cases into *test groups* as seen in the Figure 8. Each test groups exercises one dependency rule for a specific scenario. For instance, the test group *readyToArchive_90* contains generates only one test case that satisfies the dependency rule: RECEIPT = NOT IN(U,Z) AND EXTERNAL = I AND (RECALC = NULL OR RECALC = V) AND STATUS = 90. While, the test group *Possible2Rerun* is generated from a more relaxed dependency rule: RECEIPT = * AND EXTERNAL=O AND RECALC=* AND (STATUS = 90 OR 91 OR 92 OR 93 ..OR 99).

Test Scenario: Traffic is run until we have enough customs declarations in the test database. All traffic, including online, is then stopped. We check the database using DEPICT with the model in Figure 8 as input. The tester runs the archiving batch application. Then, we run DEPICT on the model again and compare the *frequency of test cases* in Figure 8 from two generated HTML reports. For instance, in Figure 9 we show a

[6] http://ssb.no

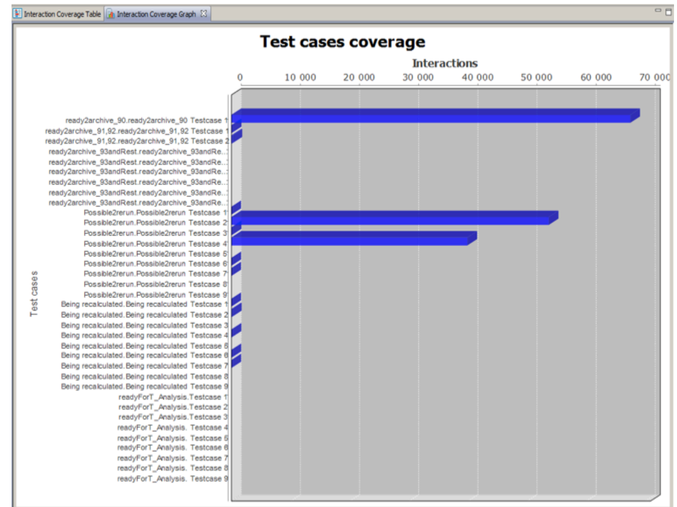

Fig. 9. Snapshot of DEPICT Monitoring Status of Archiving

snapshot of DEPICT representing number of declarations that satisfy the different test cases in the model.

If the first run exposes no data interactions (or satisfaction of test cases) for test group *ready2archive* but several interactions for the test groups *Possible2rerun* or *readyForT_analysis*, we know that there is a need to run other batch applications for recalculation and specific analysis before running a batch for archiving the declarations. Declarations cannot be archived automatically due to several reasons such as: (a) their possibility for recall for payment or review, or (b) ensuring that the content of the declarations have been sent to external entities such as the Statistical Bureau of Norway. Our approach using DEPICT allows monitoring the status of declarations before they are archived in the most optimal way. If declarations are archived before they are ready then the recall cost is relatively high as retrieval is time consuming.

VI. Lessons Learnt

This article reports an experience in inculcating *model thinking* and *combinatorial thinking* to improve testing of data-intensive systems. We focus on a part of the testing process that involves verifying test databases for data interactions. We present lessons learnt from our experience in an optimal archiving process:

Improved model thinking: Thinking about data interactions between fields cross-cutting a database schema seemed like a natural way for us to represent consistency and correctness in a test database. Our objective in this collaboration with DNCE was to validate this hypothesis. We introduced modelling data interactions using classification tree models in a robust and well-supported tool Testona at the DNCE. Testona has both a free and a commercial flavour. After one year of experimentation, the DNCE bought *five floating licenses* for the commercial version. The commercial version has support for automatic generation of test cases based on coverage criteria, such as pairwise and three-wise. Modelling data interactions also increased the readability and maintainability

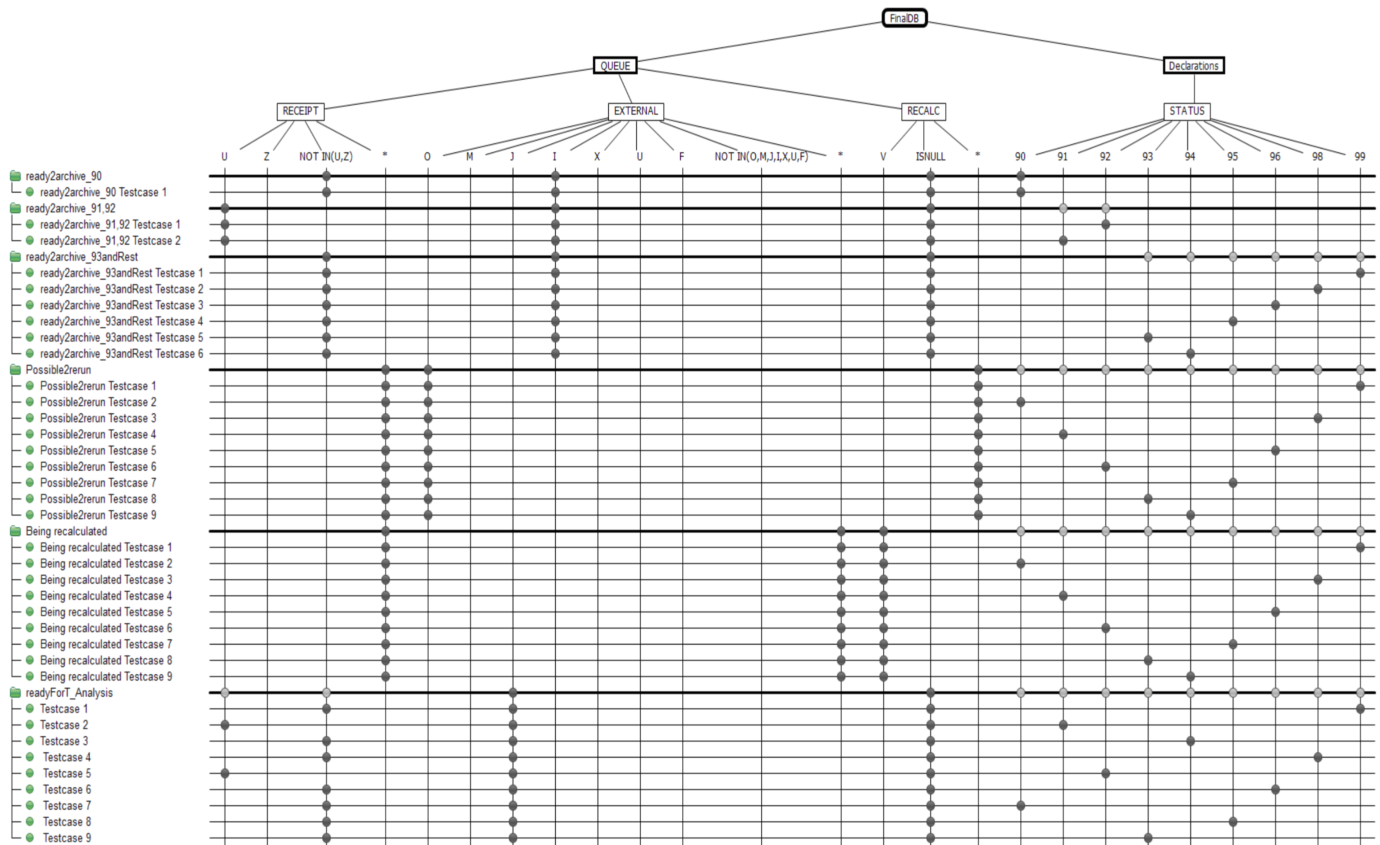

Fig. 8. Models of Combinatorial Interactions for Declaration Archival

of test scenarios compared to the state of practice (of using a test matrix in an Excel sheet or SQL queries). Currently, we are addressing the challenge of introducing modelling across the organization (DNCE), especially with testers who have not been part of the initial development process. We aim to achieve this incrementally by reuse of existing models, creation of small models, followed by use of complex features such as use of dependency rules and T-wise generation to create large models.

Advantage of associating executable meaning to models via DEPICT: Associating meaning to models of data interaction by generation of queries, their execution, and HTML report generation, immensely increased the credibility in the approach. The test managers got a first hand feel for what the test cases meant in a real artifact, which was the database FINALDB with millions of real records. The use of count or frequency in the number of times a test case was satisfied by a test database was very easy to understand for test managers and made the approach very reactive.

Very specific use of combinatorial thinking: It took test managers at least a couple of months to think about a scenario that could benefit from automatic test generation satisfying pairwise and dependency rules. Test managers often have a data interaction in mind but do not think about a full domain of data interactions and a selection of test cases based on coverage criteria. The archiving operation discussed in the paper was a good candidate for combinatorial thinking. We realize that there is to trained testers to think about combinatorics for generating a large variety of test cases using dependency rules. This can greatly improve their operational efficiency in specifying tests.

Tool adoption: Both Testona and DEPICT are used at DNCE. Our objective in the year 2015 is to produce at least twenty significantly large models for TVINN. Twelve people from DNCE are involved in the adoption efforts. There is an internal meeting every month where test managers discuss their efforts. We have a fortnightly meeting with test managers to discuss usability and new features for DEPICT. Issue reports for DEPICT are collected and addressed in a Bitbucket repository.

VII. RELATED WORK

This article addresses three principal areas of work: (a) the notion of test coverage in data-intensive systems, (b) using high-level models to specify testing intentions, and (c) combinatorial interaction techniques. We position our work in relation to these areas of work.

Test coverage: The coverage of an input domain is an important topic in testing database applications [18]. Test coverage in data intensive systems has been the subject of many studies [30], [36], [44]. These techniques are not applicable to measuring coverage in databases since they do not handle the structure of a database's complex schema. The tool proposed by Suarez *et al.* [42] measures the coverage of SQL queries without support for monitoring coverage. Halfond *et al.* [16] measure coverage of application-database interactions but do not consider the interactions between database fields. In [19],

the authors present the concept of database-aware test coverage monitoring that instruments the program and the test suite to determine how well are database entities covered. The proposed coverage monitor also captures database interactions at different levels of interaction granularity: database, relation, attribute, record, and attribute value. However, it does not provide high-level modelling of test cases as interactions. Tuya *et al.* propose a criterion that assesses the coverage of the test data in relation to the executed database queries [45]. Still, similarly to the previous approaches, it does not support modelling the test cases visually nor monitoring the coverage. There is also large body of work on generating SQL queries [2] [20] [39] [6] [31] [17] [5] [1] [50] [28] [43] [29] [38] that by construction aim to cover a database's input domain. These approaches are useful when real test databases are non-existent, or when automatically generated tests that satisfy generic constraints, such as cardinality [6], can be considered as effective. In this paper, we consider the specific scenario where real test databases are already available and need to be verified for test coverage.

Modelling test intentions: High-level specifications such as models have been used to either derive tests or simplify specification of testing intentions. Model-based testing [47] [46] is an effective approach to use behavioral models such as state machines to derive test cases. In [50], the authors use constrained queries to model database states and generate test records. Constrained queries cannot be seen as models at high-level of abstraction that significantly reduce human-effort in specifying testing intentions. In [7], the authors show that combinatorial interaction designs are very effective in constraining the input domain and consequently revealing bugs in software. We previously extended the notion of (combinatorial) interactions to represent testing intentions as data interactions in databases [37]. The notion of data interactions proposed in the paper is similar to the idea of data dependencies proposed in [11][12]. The authors propose a theoretical framework to specify *conditional functional dependencies* to improve data quality in relational databases. However, there is no mention of industry strength tool-support or modelling tools that we deem necessary for industrial impact.

Combinatorial interaction testing: Combinatorial interaction principles have a wide spectrum of applications in software testing, such as test case generation for large distributed systems [21], GUI testing [27], fault localization [51], filure diagnosis [26]. The effectiveness of combinatorial interaction techniques is based on the observation that software failures are often due to interactions between only few software parameters [21], [3], [14]. To support the application of combinatorial interaction techniques, a large number of tools has been developed, such as AETG [8], IPO [24], CATS [41], OATS [4], PICT [9]. However, our approach concerns less researched area of applying combinatorial interaction techniques for verifying the correctness of test databases.

VIII. Conclusion

We present the application of model and combinatorial thinking at DNCE using the tools TESTONA and DEPICT to verify coverage of data interactions (test cases) in a test database. The domain of data interactions and test cases are represented in a classification tree model. DEPICT connects to a test database and verifies if these test cases are covered by the database. We evaluate our approach on a *periodic archiving operation* performed at Directorate of the Norwegian Customs and Excise. Lessons learnt from the experience show that model and combinatorial thinking is useful in the industry. However, there are small challenges in having testers think about modelling domains of test cases/data interactions instead of individual test cases. Future work for the improvement of DEPICT at DNCE involves support for multiple databases for checking consistency and correctness of data interacting across databases. We are currently experimenting with our approach on other industrial databases such as databases at the Cancer Registry of Norway for modelling and verifying data quality of cancer screening data, and sensor data collected on oil pipelines at ROSEN group.

IX. Acknowledgments

We thank the Norwegian Research Council for funding our work through the Certus-SFI [7] scheme.

References

[1] IBM DB2 test data generators. http://www.ibm.com/developerworks/data/library/techarticle/dm-0706salkosuo/index.html.

[2] S. Abdul Khalek and S. Khurshid. Automated sql query generation for systematic testing of database engines. In *Proc. of the IEEE/ACM Int. Conf. on Aut. Sof. Eng*, pages 329–332. ACM, 2010.

[3] K. Bell and M. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *Inf. and Commun. Tech., 2005. Enabling Tech. for the New Knowledge Society: ITI 3rd International Conference on*, pages 221–235, Dec 2005.

[4] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of at&t pmx/starmail using oats. *AT&T Technical Journal*, 71(3):41–47, 1992.

[5] N. Bruno and S. Chaudhuri. Flexible database generators. *VLDB*, 2005.

[6] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *Knowledge and Data Engineering, IEEE Transactions on*, 18(12):1721–1725, Dec 2006.

[7] D. Cohen, S. Dalal, M. Fredman, and G. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[8] D. Cohen, S. Dalal, M. L. Fredman, and G. Patton. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, Jul 1997.

[9] Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. *In Proceedings of the 24th Pacific Northwest Software Quality Conference*, 2006.

[10] C. J. Date. *An Introduction to Database Systems*. Pearson Addison-Wesley, Boston, MA, 2004.

[11] W. Fan. Dependencies revisited for improving data quality. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 159–170, New York, NY, USA, 2008. ACM.

[12] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2):6:1–6:48, June 2008.

[13] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[14] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[15] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[7] http://certus-sfi.no

[16] W. Halfond and A. Orso. Command-form coverage for testing database applications. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 69 –80, sept. 2006.

[17] K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *Proc. of the 32Nd Int. Conf. on Very Large Data Bases*, VLDB '06, pages 1243–1246. VLDB Endowment, 2006.

[18] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *In Proc of 9th ESEC/10th FSE*, pages 98–107, 2003.

[19] G. M. Kapfhammer and M. L. Soffa. Database-aware test coverage monitoring. In *Proceedings of the 1st India software engineering conference*, ISEC '08, pages 77–86, New York, NY, USA, 2008. ACM.

[20] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proc. of the 2008 23rd IEEE/ACM Int Conf. on Automated Software Engineering*, pages 238–247, 2008.

[21] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, June 2004.

[22] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. Technical Report 1999-42, Stanford InfoLab, 1999.

[23] E. Lehmann and 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)*, pages 1–10, 2000.

[24] Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society.

[25] H. Lu, H. C. Chan, and K. K. Wei. A survey on usage of sql. *SIGMOD Rec.*, 22(4):60–65, Dec. 1993.

[26] R. Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *Community ACM*, 28(10), 1985.

[27] A. M. Memon and M. L. Soffa. Regression testing of guis. *SIGSOFT Softw. Eng. Notes*, 28(5):118–127, Sept. 2003.

[28] K. Pan, X. Wu, and T. Xie. Guided test generation for database applications via synthesized database interactions. *ACM Trans. Softw. Eng. Methodol.*, 23(2):12:1–12:27, Apr. 2014.

[29] K. Pan, X. Wu, and T. Xie. Program-input generation for testing database applications using existing database states. *Aut. Sof. Eng.*, pages 1–35, 2014.

[30] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 277 –284, may 1999.

[31] M. Poess and J. M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1045–1053. VLDB Endowment, 2004.

[32] P. Reisner. Human factors studies of database query languages: A survey and assessment. *ACM Comput. Surv.*, 13(1):13–31, Mar. 1981.

[33] E. Rogstad, L. Briand, R. Dalberg, M. Rynning, and E. Arisholm. Industrial experiences with automated regression testing of a legacy database application. In *Software Maintenance (ICSM), 2011 27th IEEE Int. Conf.e on*, pages 362 –371, sept. 2011.

[34] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. Wysiwyt testing in the spreadsheet paradigm: an empirical evaluation. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 230–239, 2000.

[35] D. Rubel. The heart of eclipse. *Queue*, 4(8):36–44, Oct. 2006.

[36] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *IEEE/ACM ASE*, ASE '07, pages 343–352, New York, NY, USA, 2007. ACM.

[37] S. Sen, J. de la Vara, A. Gotlieb, and A. Sarkar. Modelling data interaction requirements: A position paper. In *Model-Driven Requirements Engineering (MoDRE), 2013 International Workshop on*, pages 50–54, July 2013.

[38] S. Sen and A. Gotlieb. Testing a data-intensive system with generated data interactions. In C. Salinesi, M. Norrie, and s. Pastor, editors, *Advanced Inf. Systems Eng.*, volume 7908 of *Lecture Notes in Computer Science*, pages 657–671. 2013.

[39] S. Sen and A. Gotlieb. Testing a data-intensive system with generated data interactions: The norwegian customs and excise case study. In *CAISE*, Valencia, Spain, June 17-21 2013.

[40] S. Sen, C. Ieva, A. Sarkar, A. Sander, and A. Grime. Experience report: Verifying data interaction coverage to improve testing of data-intensive systems: The norwegian customs and excise case study. In *Software Reliability Eng. (ISSRE), 2014 IEEE 25th Int. Symposium on*, pages 223–234, Nov 2014.

[41] G. B. Sherwood, S. S. Martirosyan, and C. J. Colbourn. Covering arrays of higher strength from permutation vectors. *Journal of Combinatorial Designs*, 14(3):202–213, 2006.

[42] M. J. Suárez-Cabal and J. Tuya. Using an sql coverage measurement for testing database applications. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 253–262, New York, NY, USA, 2004. ACM.

[43] K. Taneja, Y. Zhang, and T. Xie. Moda: automated test generation for database applications via mock objects. In *ASE*, pages 289–292, 2010.

[44] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, July 2002.

[45] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva. Full predicate coverage for testing sql database queries. *Software Testing, Verification and Reliability*, 20(3):237–288, 2010.

[46] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[47] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report 04/06, New Zealand, April.

[48] C. Welty and D. W. Stemple. Human factors comparison of a procedural and a nonprocedural query language. *ACM Tran. Dat. Sys.*, 6(4):626–649, 1981.

[49] S. White, Cattell, Fisher, and Hamilton. *JDBC API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[50] D. Willmor and S. M. Embury. An intensional approach to the specification of test cases for database applications. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 102–111, New York, NY, USA, 2006. ACM.

[51] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, Jan 2006.

[52] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. Automatic test generation using checkpoint encoding and antirandom testing. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, pages 84–, Washington, DC, USA, 1997. IEEE Computer Society.