Test Selection based on Data Interactions in Data-intensive Systems

Sagar Sen, Carlo Ieva, Dusica Marijan, Arnab Sarkar, Arnaud Gotlieb
Certus V&V Center
Simula Research Laboratory
Oslo, Norway
Email: {sagar,carlo,dusica,arnaud}@simula.no, arnabcricket@gmail.com

***Abstract*—Testing data-intensive systems is paramount to increase our reliance on information in e-governance, scientific/medical research, and social networks. Common practice to test these systems is by using a *live production database*. This testing approach is space and time inefficient and lacks clarity about what test cases or scenarios are covered. In this paper, we leverage classification tree modelling to specify desired test cases as *data interactions* between a set of fields across multiple tables of an existing database. Our methodology and tool, DEPICT, uses test case specifications in classification tree models to (a) automatically derive a *spanning tree* representing a relationship between any set of fields for any given database schema (b) generates queries to create an efficient inner join between related tables in the spanning tree (c) extract records from various tables that satisfy data interactions in the classification tree model (d) discovers holes or unsatisfied test cases in the test databases. We perform experiments to show that our approach is fast and scalable to extract test databases. Our experiments are based on selecting test databases from 8000 declarations for 60,000 items from the Norwegian Customs and Excise information system TVINN.**

I. INTRODUCTION

Data-intensive software systems are increasingly prominent in driving global processes such as scientific and medical research, e-governance, and social networking. Large amounts of data is collected, processed, and stored by these systems in *databases*. For example, the Norwegian Customs and Excise department (NCE) uses the TVINN system to processes about 30,000 declarations a day as shown in Figure 1 (a). TVINN stores validated transactional information such as declarations in a central production database. It processes incoming declarations to verify their conformance to well-formedness rules, customs laws and regulations before accepting a declaration in the database. This scenario is prevalent in many data-intensive software systems dealing with *transaction data* which comprises semi-structured/structured data in medium/high volume. Testing these data-intensive information systems is the subject of this paper.

The common practice in testing a data-intensive system entails the usage of a *test database* obtained by selectively querying a production database of *live and recent transactions*. The records in the test database are processed by a database application to reveal faults in business rules. They are also used for regression testing between different versions of the database application. In this paper, we address several challenges not addressed by common testing practice (1) There exists no high-level view or model of what testing-specific data a test database contains (2) The test database does not guarantee any form of test adequacy [4] such as an input domain coverage criteria (3) Is the size of the test database minimal for increased testing efficiency?

In this paper, we present a methodology and a tool suite DEPICT to address the above challenges. We first propose the use of *classification tree models*[6] to represent test cases at a graphical and high-level of abstraction. We use classification trees to specify *interactions between database field values* to address challenge 1. DEPICT surgically mines the database for coverage of test cases in the classification tree model. DEPICT first creates a spanning tree to find a path of interaction between different tables specified in the classification tree model. DEPICT transforms test cases to queries that extract a view (from a inner join of the tables in the spanning tree) and counts occurrences of interactions. The results are presented using graph representations that help a tester evaluate his test database for minimality and interaction coverage. The tester can then complete the database or reduce its size by removing redundant test cases hence addressing challenges 2 and 3.

We evaluate our approach through experiments on test databases from the Norwegian Customs and Excise department. We first present an experiment on a test database containing 8400 real or live customs declarations to check for interaction coverage. The experiment clearly illustrates the *non-satisfaction of combinatorial interaction coverage* in a test database with live declarations. In another experiment, we use a synthetic test database to demonstrate (a) combinatorial interaction coverage (b) that the approach is scalable and of constant time complexity taking only 1 ms on an average to synthesize a single interaction coverage query, execution of query, and visualization of results.
We summarize our contributions as follows:

Contribution 1: We present a methodology and a tool DEPICT to model, extract and visualise interaction coverage of test cases in a data-intensive systems
Contribution 2: We demonstrate through experiments on an industrial case study (NCE) that our methodology can help ensure combinatorial interaction coverage in a scalable manner

The paper is organized as follows. In Section II, we present an industrial case study from NCE as a running example in the paper. In Section III, we present our methodology and tool DEPICT. We present results from experiments in Section IV. Section V discusses the related work, while we conclude with a summary of our experience in Section VI.

II. INDUSTRIAL CASE STUDIES

We describe our industrial case study from the NCE as illustrated in Figure 1(a). As mentioned above, the system under study is TVINN. An overview of TVINN process flow is presented in Figure 1(a) and an official description is available on its website [1].

Customs officers and industries associated with import and export create declarations at Norwegian ports of entry. These declarations are encapsulated in the EDIFACT standard for business communication. A declaration is encapsulated as an EDIFACT CUSDEC message. These messages are sent to TVINN's central server where they are processed by a sophisticated batch application called EMIL. EMIL parses EDIFACT messages and verifies them against well-formedness rules. It then verifies if the declared amount is accurately computed based on a statistical value for an item. These rules depend on numerous factors such as (a) 260 countries of origin divided in 88 country groups. (b) over 160 currencies. (c) around 900 tax code groups, and (d) a list of more than 10,000 items. A declaration can be categorized into six different categories based on EMIL's computation, the simplest categories being *complete* and *reject*. The response from TVINN is sent back as an EDIFACT CUSRES message to customs officer or industry.

Rules 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. For instance, a customs officer could decide to thoroughly check 20 trucks coming from a nation X in civil war and he/she would create a rule to check all the trucks from the nation for the following three hours. These kinds of rules are called *mask control* and will disappear 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 executing a subset of a large number of *live declarations* as test cases. However, using these test cases present four important problems:

No Coverage Guarantee: Test cases obtained from live declarations, cover a realistic subset of the database's domain (set of all possible combination of values in fields and tables of a database). However, they often do not cover combinations of values that are very rare or exceptional. Test cases often need to model such *holes* in real data.

Very Large Set of Test Records: Accumulating information from live transactions can easily give rise to an ever-growing set of data records. Many of these records share similarities and hence are redundant for the purpose of testing. Cost-effective testing will require a selection of a minimal set of test cases. A minimal set will also have modest time and space requirements for testing efforts such as nightly tests.

Confidentiality: Governments/enterprises involving financial transactions or military data for instance have stringent confidentiality agreements with their clients. Therefore, it is often not possible for them to outsource their testing efforts to external agencies.

Constantly Changing Rules: Test cases have a lifetime and need to be discarded. For instance, in the NCE system changes when sanctions are imposed on countries or significant changes happen in currency exchange rates. Legacy test cases may not be used anymore to test the evolved system.

In this paper, we address the **specific problem of test coverage in test databases**.

III. METHODOLOGY

In this section, we present a methodology for model-driven interaction testing. The foundation for the approach are presented in Section III-A. The overview of our methodology is shown in Figure 3. In Section III-B, we describe modelling interactions that is used in our methodology. We present the different steps in our methodology in Section III-C. In Section III-D, we present how DEPICT is adopted at NCE.

A. Foundations

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 [2]. 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 Structured Query Language (SQL) queries. We use queries to create views and to select and count number of records. In this paper, we use the database schema, shown in Figure 1(b) for the NCE case study.

[1] https://fortolling.toll.no/Tvinn-Internett

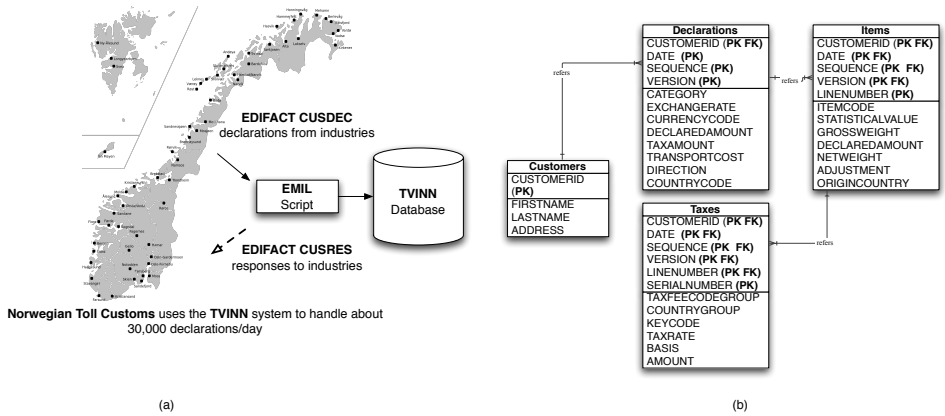

Fig. 1. (a) Norwegian Toll Customs Industrial Case Study (b) Common Database Schema at Norwegian Toll Customs in Crow-Foot Notation

2) Classification Tree Model: We use the *classification tree models* to graphically represent test cases as interactions between database field values. A specific set of test cases is said to satisfy the combinatorial interaction coverage criteria [1] when it covers all T-wise interactions between a set of database fields. The most effective interaction coverage criteria is 2-wise or pairwise. The models are created using the tool CTE-XL tool[6]. CTE-XL is an editor based on the classification-tree method, as an approach to category-partition validation that uses a descriptive tree notation. This tool can scale up to the complexity of input domains such as the one of the Norwegian Tax Department [8]. CTE-XL has the notion of *compositions, classifications and classes* to model variability in a database field values. The variability is typically modelled as a tree. CTE-XL also allows the specification of *dependency rules* as boolean constraints to constrain selection of classes across the tree. It provides all features necessary to model a constrained domain. In Figure 2, the top-level composition is the name of the database, the second-level compositions are table names, the third-level classifications are field names and finally in the fourth-level we have classes representing values that go into fields. Data interaction requirements can be specified manually in CTE-XL by clicking on a button for each field value in a requirement. We specify *data interaction requirements* in CTE-XL as *test cases*. CTE-XL can also be used to automatically generate interaction requirements that cover all pair-wise or three-wise interactions between classes of choice. For instance, in Figure 2, we present all pair-wise interactions between five declaration categories and five tax fee codes such as VAT.

B. Modelling Interactions

We assume that a modeller is knowledgeable about the database schema of the information system for which data interaction requirements are specified. The modeller specifies data interaction requirements as a classification tree model. An example is illustrated in Figure 2. The different elements of the classification tree model for data interactions are 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 2, this is represented by the composition TollCustomsDemo.

Compositions for Tables: The root composition can contain several compositions representing identifiers for tables. In Figure 2, this is represented by the compositions Declarations and Items from the schema shown in Figure 1(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 2, we use the fields Category and ItemCode. All or only a subset of fields for a table maybe be specified in the model.

Classes for Field Values: The different values for fields are classes in the model. For instance, in Figure 2, the fields values MA and FO are associated with the classification for the field Category. Field values are unique and an interaction can have exactly one possible field value.

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 2, pairwise.TestCase1 represents the interaction {MA,MV_2}. This is the interaction between a tax fee code group for value added tax (VAT) and manual processing of a declaration. One may envisage the use of such an interaction to generate customs rules. Interactions in CTE-XL can either be generated automatically such that all pairs or three-wise interactions between two or three classes are covered. In Figure 2, 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.

The tool CTE-XL also allows specification of additional boolean constraints between classes, compositions, and classifications to limit the number of interactions or test cases.

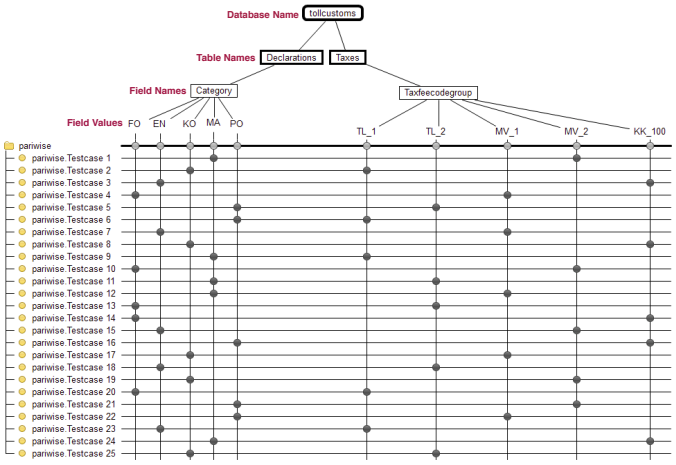

Fig. 2. Classification Tree Diagram Model of Test Cases

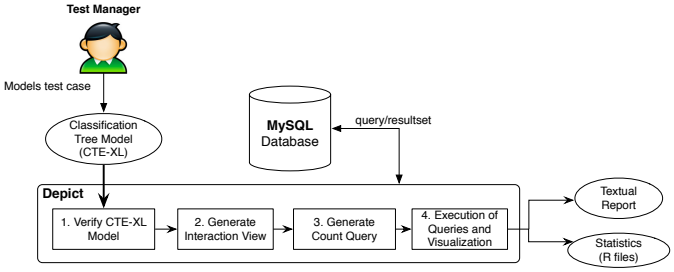

Fig. 3. Methodology for Modelling and Visualizing Interaction Coverage

C. Methodology Description

The methodology implemented in DEPICT is illustrated in Figure 3.

Step 1, DEPICT verifies that the input CTE-XL model 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 such as MySQL.

Step 2, DEPICT transforms a CTE-XL model to graph of internal database schema and SQL queries to extract data from it. DEPICT first creates an internal *spanning tree* between tables specified in the CTE-XL model. In the second step DEPICT creates an SQL view for interactions in the spanning tree. For instance, given two fields we present the view created in Listing 1. The principal challenge for the generation of this query is to ensure that the database fields indeed *interact*. The transformation leads to a query ensures that keys between tables Table1, Table2, ..TableN match so that the collected records in the view are *interacting*. Field values in a database do not interact if they are not associated by matching key values across the database. For instance, a valid interaction occurs between a declaration category field and a customer id field only when the value of customer id as primary key has a value identical to a matching foreign key in the declaration table. A declaration record must belong to a customer record.

Step 3, DEPICT generates SQL count queries as shown in Listing 2 to count all T-wise interactions/combinations for each test case. For instance, DEPICT counts all pairwise combinations $(0,0),(0,1),(1,0),(1,1)$ of two field values.

Step 4, DEPICT runs the created view and count queries to obtain the frequency of interactions in the database. A *frequency of zero* implies that the test case was not covered by the test database. While a very high frequency indicates redundancy of test cases hence requiring test set minimization. We also generate graphs in the statistical language R and Google Charts to compactly represent interaction coverage and time consumption. We use the bar-chart and the spider diagram to provide two different views of combinatorial interaction coverage.

```
CREATE VIEW PAIR_INTERACTION AS SELECT FIELD1, FIELD2 FROM
TABLE1 LEFT JOIN TABLE2
ON (TABLE1.Key1=TABLE2.Key1) AND (TABLE1.Key2=TABLE2.Key2);
```

Listing 1. Create View Query to Accumulate Interactions

```
SELECT COUNT(*) FROM PAIR_INTERACTION WHERE FIELD1<>"VALUE1"
and KEYCODE=<>"VALUE2";
SELECT COUNT(*) FROM PAIR_INTERACTION WHERE FIELD1<>"VALUE1"
and KEYCODE="VALUE2";
SELECT COUNT(*) FROM PAIR_INTERACTION WHERE FIELD1="VALUE1"
and KEYCODE=<>"VALUE2";
SELECT COUNT(*) FROM PAIR_INTERACTION WHERE FIELD1="VALUE1"
and KEYCODE="VALUE2";
```

Listing 2. Select Count Query to Count Interactions

D. Implementation and Industrial Adoption

DEPICT is implemented in pure Java and is available for download and use on a website [2]. We report on how our methodology is being adopted at the NCE. The TVINN system at NCE contains about 138 tables which include 121 tables (with 1250 fields) in TVINN itself and 17 tables (with 182 fields) in the POST database (after processing of a declaration). The databases TVINN and POST are highly dynamic and a snapshot of these tables can contain up to 25 million records. We present a simplified subset of TVINN's schema in Figure 1(b). For the moment test managers at TVINN export existing records to a test database conforming to 1(b). They then use CTE-XL to specify test cases that are verified by DEPICT on the test database. In case a test cases is not covered, test managers prefer to manually create a new entry on the TVINN system to cover the test case. Test managers report that the high-level model helped them understand the intention of each test case and maintain traceability with the database, as DEPICT provides the exact location of the desired interaction. The lightweight standalone tool DEPICT is designed to be scalable to the entire TVINN database.

IV. EXPERIMENTAL EVALUATION

We present two experiments for the Norwegian Customs and Excise case study to illustrate (a) the problem in common testing practice (b) the advantage of a test database that covers all pairwise interactions between certain field values.

[2]https://sites.google.com/a/simula.no/depict/

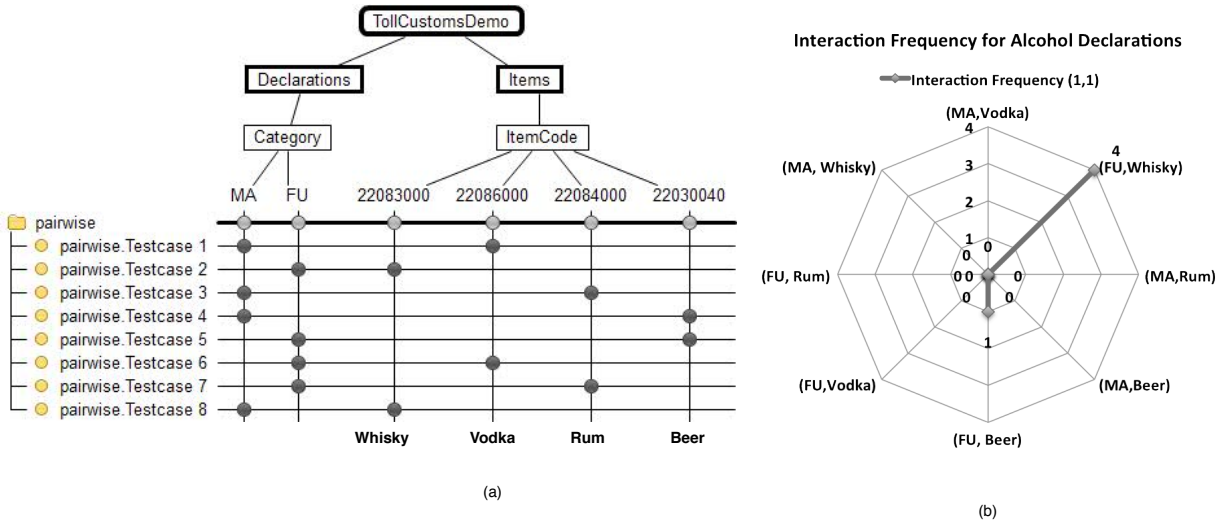

Fig. 4. (a) Test Cases for Alcohol Declarations covering Pairwise Coverage (b) Spider Diagram of Coverage in 8400 real declarations

A. Illustrative Experiment

The common practice of testing involves the usage of a large number of live customs declarations in a test database to test the database application at NCE. Consider the CTE-XL model of Figure 4 (a). The model encodes test cases that cover all pairwise interactions between (a) four types of alcohol Whisky, Vodka, Rum, and Beer with their respective ItemCodes and (b) manual inspection (MA) or processed declarations (FU). We use a real test database of 8400 declarations from a given day. Using DEPICT, we summarize the results in the spider diagram of Figure 4 (b). In the figure most of test cases did not exist in the test database except four instances of (FU,Whisky) and one instance of (FU, Beer). This simple experiment illustrate that the common practice of simply using a large test database does not guarantee a combinatorial interaction coverage criteria such as pairwise.

B. Evaluating a Complete Test Database

We perform a second experiment using DEPICT to verify interaction coverage in a test database [3] for the model shown in Figure 2. The model specifies 25 test cases covering all pairwise interactions between a set of tax fee code groups and declaration categories. The test database was synthetically generated using the approach presented by the authors in [10]. The goal of the experimental evaluation is to answer two questions: Q1) What is the interaction coverage achieved by the test database? and Q2) What is time efficiency for interaction coverage in a realistic setting?

The experiments were run on a laptop with Pentium(R) Dual-Core CPU E5300 @2.60GHz 2.60GHz and 2GB of RAM.

1) Test Coverage: DEPICT is run for the model in Figure 2. It outputs a bar-graph, as shown in Figure 5 representing interaction coverage in the database. Since, the test database was synthetically generated it contains all test cases at different frequencies. The x-axis of the plot represents the 25 test cases. The y-axis presents the frequency of interactions observed in the database for each test case. In Figure 5, we observe that the synthetic test database covers all pairwise four variations $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ of 25 test cases with non-zero interaction frequency.

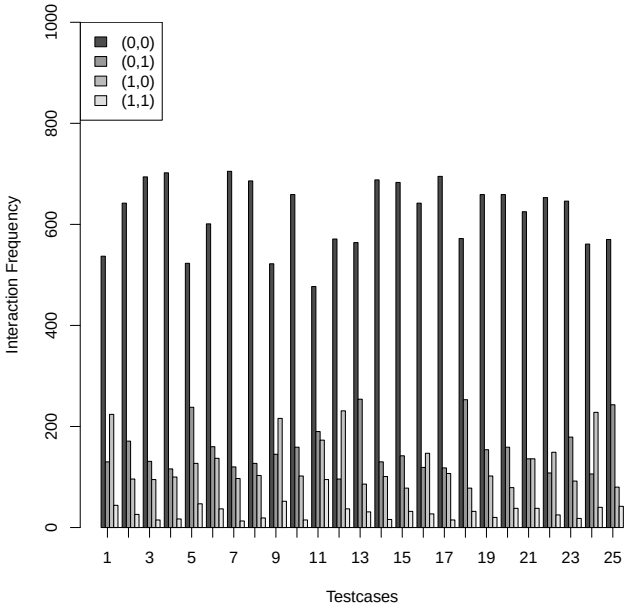

Fig. 5. Interaction Coverage for Pairwise Test Cases in Synthetic Test Database

The bar-graph of Figure 5 and the spider plot of Figure 4 (b) give a compact representation of covered and missing interactions in a test database.

[3]https://sites.google.com/a/simula.no/dbtwise/

2) Time Efficiency: How much time does it take to detect if test cases are covered by a database? We observe that it took an average of 1 ms to detect an interaction in database of about 1000 records. This includes the time to create a view for interactions, executing a query to count frequency of the interactions, and return of report via JDBC to the user. This indicates a constant time complexity $\mathcal{O}(1)$. We also execute DEPICT 30,000 times to obtain interaction coverage in a total time of 8 seconds. The number 30,000 is also the number of live declarations made per day within TVINN.

V. Related Work

Input domain coverage is an important topic in testing database applications [4]. In this paper, we apply combinatorial interaction coverage [1] to help develop test databases for database applications.

Test coverage in data intensive systems has been the subject of many studies [7], [9], [12]. However, 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 [11] measures the coverage of SQL queries, it does not support coverage monitoring. The work proposed by Halfond [3] measures the coverage of defined testing requirements for database commands. Halfond measures the coverage of application-database interactions and does not consider the interactions between database fields. In [5], 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. From the standpoint of our industrial partner, NCE, it was essential for them to have a high-level view of a test case to understand testing intention. Tuya proposes a criterion that assesses the coverage of the test data in relation to the executed database queries [13]. Still, similarly to the previous approaches, it does not support modelling the test cases visually nor monitoring the coverage.

VI. Conclusion

In this paper, we present a methodology and tool DEPICT to detect coverage of test cases in a database. We perform experiments for an industrial case study from the Norwegian Customs and Excise department. The experiments reveal the interaction coverage in the database in a very time efficient manner (8 seconds for 30,000 test cases). Our methodology and DEPICT is currently being adopted at the customs department. The test managers at the customs department state that the high-level view of test cases greatly enhances their understanding of their intention. They also appreciate the traceability of a test case /interaction to a concrete record in the database provide by DEPICT. We foresee, (a) improvement of the modelling tool to be able to specify test oracles with regard to desired properties in a database (b) to explore visualization approaches to be able to compactly represent test case coverage for very large number of records (c) DEPICT can be seen as a metric module for the degree of coverage. These metrics can be used by test set selection, minimization, and prioritization algorithms to create minimal test databases.

References

[1] 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.

[2] C. J. Date. *An Introduction to Database Systems*. Pearson Addison-Wesley, Boston, MA, 8. edition, 2004.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] E. Rogstad, L. Briand, R. Dalberg, M. Rynning, and E. Arisholm. Industrial experiences with automated regression testing of a legacy database application. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 362–371, Washington, DC, USA, 2011. IEEE Computer Society.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.