Abstract—Testing data-intensive systems is paramount to increase our reliance on information processed in e-governance, scientific/medical research, and social networks. A common practice in the industrial testing process is to use test databases copied from live production streams to test functionality of complex database applications that manage well-formedness of data and its adherence to business rules in these systems. This practice is often based on the assumption that the test database adequately covers realistic scenarios to test, hopefully, all functionality in these applications. There is a need to systematically evaluate this assumption. We present a tool-supported method to model realistic scenarios and verify whether copied test databases actually cover them and consequently facilitate adequate testing. We conceptualize realistic scenarios as data interactions between fields cross-cutting a complex database schema and model them as test cases in a classification tree model. We present a human-in-the-loop tool, DEPICT, that uses the classification tree model as input to (a) facilitate interactive selection of a connected subgraph from often many possible paths of interactions between tables specified in the model (b) automatically generate SQL queries to create an inner join between tables in the connected subgraph (c) extract records from the join and generate a visual report of satisfied and unsatisfied interactions hence quantifying test adequacy of the test database. We report our experience as a qualitative evaluation of approach and with a large industrial database from the Norwegian Customs and Excise information system TVINN featuring large and complex databases with millions of records.

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 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.

The typical process to rapidly and effectively test batch applications (including regression testing [24]) on data-intensive systems involves usage of input test databases regularly copied from the live transaction processing stream. Using test databases is based on the assumption that data from live transactions represents realistic scenarios. The realistic scenarios correspond to patterns found in test databases that are expected to either demonstrate correctness of application functionality or uncover bugs in the way transactions were processed by batch applications. For instance, testing a customs regulation or business rule in the TVINN system for value added tax (VAT) on alcohol such as whisky will require a test database with customs declarations for imports on a specific type of alcohol, whisky, and specific kind of tax, VAT. There is often a high probability that declarations coming into TVINN have exercised and consequently tested a large number of business rules. However, despite a high practical reliance on such test databases there exists very few systematic approaches to verify their adequacy for testing. This is the problem area we address in the overall testing process. We believe that verifying test databases in long-running data-intensive systems with medium/high volume of transactions will ensure adequate coverage[13] to test batch application. Verification will also make the overall testing process efficient as manual testing (such as creating specific declarations by customs personnel) can be limited to those cases that have not been covered by data from live streams. Therefore, we ask, how can we automate and simplify steps in the verification of a test database to improve testing of data-intensive systems?
This is the general question that intrigues us and the subject of this paper.

**Existing approaches:** An obvious approach to verify test databases would be to create and execute complex SQL queries representing realistic scenarios. This approach is often tedious, error-prone [23], database technology specific, fails to capture high-level testing intentions much needed for documentation, and consequently hard to maintain [16]. Moreover, studies assessing human factors have shown that it is not easy to create such queries, especially when multiple table joins are involved [38] [23] [18]. 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 [25]. However, documenting testing intentions in a spreadsheet does not associate clear computable meaning to the tests making them ambiguous and a challenge to maintain. Synthesis of databases to test data-intensive systems is an approach widely explored with several recent approaches [19] [33] [20] [29]. Having automatically generated databases readily available can be an effective approach to test data-intensive systems under development or freshly deployed. However, in long-running data-intensive systems such as TVINN (25 years) there is already a large repository of realistic declarations to help test the system for most cases. The general consensus at the directorate of customs and excise of Norway is that manual creation or automatic generation test databases must be limited to cases not already covered by real data or automation involving migration of test databases to the evolving system. This industrial need hence helped us scope our focus to verify coverage of test databases in the global testing process of a data-intensive system.

**Contributions:**

**Test Cases as Data Interactions:** We propose 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 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[28] using the *classification tree modelling* formalism [10] and tool CTE-XL[17]. The classification tree primarily models a domain of data interactions possible between fields of a test database. A *test case* in the classification tree model represents an interaction between classes (exact values or domains of values) of one (self-referential), two or more of these fields that are either simultaneously desirable or undesirable in a test database. Representing test cases as data interactions is based on the idea that different fields cross-cutting a relational database schema have classes that interact or are *inter-dependent* (due to operations in a black-box application) and either (a) must *exist* together in the database for a fixed number of times (b) *not exist* together at all. Testing with interactions and combinations is not a new idea in software testing [9] [42]. 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.

**Verification of Data Interactions:** How can we verify if these test cases representing data interactions are in fact covered in a test database? The principal contribution of this paper is a human-in-the-loop method implemented in a tool DEPICT to answer this question. When we model a data interaction between classes of several fields in a potentially complex database schema there may be many ways to establish an interaction between them. Therefore, 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) (b) surrogate and self-referential relationships based on type matching between one or more fields between two tables. This connected subgraph is not necessarily the smallest connected subgraph as often there are many possible paths between fields of tables in a complex database schema. Selecting one of different possible connected subgraphs is *domain-specific* and is best performed by a human expert in the loop using the interactive interface. 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. This report gives comprehensive feedback to testers about the quality of their test database with regard to covering modelled realistic scenarios. Having this information helps testers manage testing redundancy and reduce effort by limiting manual creation of tests (such as customs declarations) to only uncovered realistic scenarios. The report provides confidence about the coverage of a test database accrued from live transactions which is considered to be an irreplaceable testing asset (by random generation for example) in the industry. Moreover, we observe that the classification tree modelling formalism with test cases is a graphical and intuitive approach to document tests as data interactions and easily adopted by practitioners. DEPICT is implemented as a standalone Eclipse [26] rich-client platform application and uses a non-vendor specific subset of JDBC [39] for database connectivity rendering the tool database technology agnostic. We evaluate our method and tool DEPICT on both a simplified running example and industrial test databases from the Directorate of Norwegian Customs and Excise. The industrial test databases have large and complex database schemas containing millions of real records.

The paper is organized as follows. In Section II, we describe both a simplified running case study and industrial case studies. In Section III, we present modelling data interactions with classification tree models. Section IV describes the method
and associated tool, DEPICT to verify test databases using classification tree models as input. We provide a qualitative evaluation of our approach in Section V. Section VI discusses the related work and places our contribution in the body of knowledge. We conclude in Section VII.

II. CASE STUDY

TVINN is a Customs information system for business and trade developed by and for the Norwegian Customs as shown 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 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 even greater 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 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 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 execute 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 verifying data interaction coverage with 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

2http://www.unece.org/trade/untidid/welcome.html
the paper. In Section II-B, we describe the complex 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 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 tables in TEMPDB are MHEAD (Message Head) 
and MDAT (Message Data).

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 (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.

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 [28]. Data interactions are modelled on a relational database 
schema. We briefly describe a database schema in the fol-
lowing Section III-1. We use the classification tree modelling 
formalism to model data interactions as presented in Section 
III-2.

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 [6]. 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 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 CTE-XL[17] to model clas-
sification tree models (CTM). CTMs have been used to model 
complex input domains such as the one of the Norwegian Tax 
Department [24]. 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 
3(a) and (b) 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 (a), this is represented by the composition TollCustomsDemo.

**Compositions for Tables:** The root composition can contain several compositions representing identifiers for tables. In Figure 3 (a), 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 (a), 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 (a), 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. We present an example with special classes in Figure 3(b) without test cases.

**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(a), pairwise_TestCase1 represents the interaction {MA, 22086000}. This is the interaction between a declaration category for manual processing of an import and for itemcode 22086000 for Vodka. 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 3(a), 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 a small number of test cases with very specialized testing intent. This also renders the model-based documentation more comprehensible.

The tool CTE-XL also 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. This topic is out of the scope of this paper and we suggest that the reader acquires the documentation for the professional version of CTE-XL. A free version of CTE-XL is available which was good enough for the modelling tasks in this article as we did not require the use of dependency rules and automatic generation of test cases.

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

**IV. METHOD**

Given a model of data interactions as input our method computes the satisfaction or non-satisfaction of these interactions

![Fig. 3. (a) Classification Tree Model for Alcohol Imports (b) Classification Tree Model with Special Classes for Domains](image-url)
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. The global process is out of the scope of this paper. 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 (a). We use the tool CTE-XL [17] to create the model. The CTM is then imported into D3. We use the tool CTE-XL [17] to create the model. 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 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 [39] 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 [39]. 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 (a). 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 to display, perform automatic layout, and enable

**Human Inputs**

![Diagram of DEPICT system](http://jung.sourceforge.net)

![Diagram of DEPICT system](http://jung.sourceforge.net)

**Fig. 5.** Verifying Interaction Coverage

**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 field 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 don’t exist. In that case we are able to fill the gap creating a surrogate rel.

The principal task of a human-expert is to select edges from potentially several possibilities to create a connected subgraph that is meaningful. We present such an example in the industrial Scenario 1 in Section V.

**Step 4:** The connected subgraph between all interacting nodes is used by DEPICT to generate SQL queries to create an interaction table for each test case. For instance, we generate 8 interaction tables for the model in Figure 3(a). The query to generate an interaction table for the test case pairwise: Testcase1 in Figure 3(a) 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

---

3http://jung.sourceforge.net
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.

```sql
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.data AS f_6, items.sequence AS f_7, items.version AS f_8, items.linumber AS f_9 FROM declarations JOIN items ON declarations.customerid=items.customerid 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 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 if a 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\(^4\) to display the count for each test case as shown in Figure 7 for the CTM in Figure 3(a). The report gives instant feedback to a tester about interactions that are missing in a test database, and the elapsed time to query the test database. DEPICT also generates a bar graph produced using JFreeChart\(^4\) to display the count for each test case as shown in Figure 7 for the CTM in Figure 3(a). The report gives instant feedback to a tester about interactions that are missing in a test database, and the elapsed time to query the test database.

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. INDUSTRIAL EVALUATION

We evaluate DEPICT, on the industrial test databases FINALDB and TEMPDB as described in Section II-B. We perform a qualitative case study based evaluation [41] to address the following research questions:

**RQ1:** Can modelling data interactions effectively represent realistic scenarios in test databases?

**RQ2:** What are the features of human-in-the-loop interactive interface to facilitate accurate specification of data interactions?

**RQ3:** Is DEPICT time efficient and scalable in terms of query execution and report generation?

**Scenario 1:** The first scenario presents a simple example to illustrate some of the features in DEPICT. Some customs declarations specifies legal documents for either import (I) or export (E). Such a document must belong to the legal owner of the declaration.

**Solution with Classification Tree Modelling and DEPICT:** Our objective is to verify a test database for presence of declarations with legal documents for the same legal owner. The presence of such declarations ensures testing of business rules pertaining to declarations and their legal documents. Nevertheless, testing these business rules is out of the scope of this paper.

First, we model the case in a CTM as shown in Figure 8. The CTM for the test database DB models the interaction between two tables DOCUMENT and DECLARATIONS. The fields DOCDIR for DOCUMENT and DECDIR for DECLARATIONS have two possible classes I (import) and E (export).

The different test cases in Figure 8 represent all possible combinations of import and export of documents and declarations. This simple model addresses **RQ1**, by demonstrating the ease with which the CTM represents all possibilities pertaining to Scenario 1. The same test cases can be represented as SQL queries as shown in Column 5, Figure 10 that are long and hard to read.

Second, we import the CTM into the tool and extract the graph of the database schema and select a connected subgraph as shown in Figure 9. At this point an intervention from a human expert is necessary in order to specify the correct association between the nodes for table DECLARATIONS and DOCUMENT. The interaction between DECLARATIONS and DOCUMENT is via the node CUSTOMERS. There are three associations between DECLARATIONS and CUSTOMERS with three different semantics (as shown in the zoom-in of Figure
9): 

- The legal owner of the declaration
- The declarant (the importer)
- The payer

In this case DEPICT allows the domain expert to choose the correct association which is legal owner. Once, a connected subgraph is created DEPICT generates SQL queries to compute the number of occurrences of each interaction. This interactive interface in DEPICT addresses RQ2 on the simplicity with which interactions can be specified. DEPICT allows moving nodes and edges, zooming-in/zooming-out into the graph of the schema, and picking one out of three possible edges for Scenario 1 and consequently the validation connectedness of the selected subgraph.

Finally we address RQ3 by generating a visual HTML report as shown in Figure 10. The HTML report can be exported from DEPICT and be used as documentation for coverage of test cases/data interactions. The report shows the id of a test case, its name, the number of times the test case is satisfied in the database, a human readable version of the interaction, a generated SQL query that can be used separately to extract records for a test case, and the elapsed/response time for each query. The elapsed times using our DEPICT demonstrates its reasonable responsiveness on very large databases for Scenario 1. For instance, the maximum elapsed time to count the number of occurrences in about 2.5 million customs declarations is about 1451 ms as shown in Figure 10. A small CTM such as in Figure 8 shows reasonably low response time even for very large databases. However, performance is likely to deteriorate due to joins between several tables with several million records. We believe that constructing multiple small models representing specific testing intentions is a good practice to maintain understandability of tests.

**Scenario 2:** Every incoming customs declaration CUSDEC EDI message is validated by a batch process and a CUSRES EDI message is sent back to the declarant as shown in Figure 11. If a declaration is rejected then it is sent back with a fault code to the declarant. The CUSRES message is parsed and stored in the TEMPDB database as described in Section II-B. This database TEMPDB also stores the incoming declaration CUSDEC. The contents of both CUSDEC and CUSRES messages is stored in tables MHEAD for message header and MDATA for message data. The testing intention is to verify if a CUSRES with a given fault code only has been sent if the incoming CUSDEC has a certain CATEGORY of either EN,PO,MA or * (don’t care). For instance, a CUSRES with a fault code 736 should only be found if its corresponding incoming CUSDEC has a VERSION greater than 1 and the CATEGORY MA and the previous version of the CUSDEC has not yet been processed.

This testing intention can be modelled in a CTM as shown in Figure 12. First, we can take a look at some special and powerful modelling features derived from industrial needs. The class REC(CUSRES,CUSDEC) which is encircled indicates the self-referential relationship that the field MTYPE (message type) can have two values CUSRES and CUSDEC. The sequence of CUSRES and CUSDEC in the class field of the CTM helps specify expected values for other fields in the same table for either CUSRES (left) or CUSDEC (right). For instance, if the MTYPE is CUSDEC then we expect the value of VERSION to be equal or greater than 2 as shown in the class REC(*,RNG(2,)). The class LK(%ERC+736%) of the field MTEXT (message text) refers to a pattern in a possibly very long message. The fault code number 736 is present in a text sequence preceded by the identifier ERC. This pattern matching clause helps DEPICT generate an SQL query to select CUSRES messages with fault code 736. We recall that Figure 4, presents more detail about the special clauses that can be used for classes in the CTM. The data interaction for the highlighted test case of Figure 12 represents
the testing intention that a declaration has to be processed manually (class MA for PCATE) when VERSION is greater than 2 for the incoming customs declaration CUSDEC and the fault code (found in MTEXT) in the outgoing CUSRES is 736. The CTM concisely represents the testing intentions which otherwise are complex and significantly less human understandable SQL queries as shown in Column 5, Figure 14, hence, addressing RQ1.

In Scenario 2, the tables MHEAD and MDATA have no prior relationships and are singleton as shown in Figure 13. In this example, we illustrate how DEPICT allows the creation of custom or surrogate relationships. DEPICT allows the specification of a self-referential relationship for the message head table MHEAD as shown in Figure 13. This is necessary since CUSDEC and CUSRES messages are both stored in the same table MHEAD. This is a very common design pattern in relational database schemas of many industrial information systems. DEPICT also allows creation of surrogate relationship between two different tables that have no prior relationship. This is done by interactively selecting an ordered sequence of fields from each table, MHEAD and MDATA in this case, that have matching types. DEPICT automatically verifies the human selection of fields for a type match and does not proceed unless the selection is correct-by-construction. A traditional approach would be in two steps: (a) to query SQL metadata to find fields that type match followed by (b) creation of large SQL queries for inner joins on these fields. DEPICT, drastically simplifies this effort, favourably answering RQ2.

The relationships are checked by DEPICT generation of SQL queries for data interaction tables as shown in Column 5, Figure 14. The results show that the test database contains a few holes (indicated by highlighted boxes in Column 1, and a bar with value zero in the coverage bar-graph). For instance, in 2.5 million declarations, there were no records for fault code 736 with VERSION greater than 2, and none with fault code 450. This indicates that the test database TempDB copied from a live transaction stream at a given point in time did not have any CUSDEC messages messages failing the check which would have returned fault code 736 or 450 making the database inadequate for testing certain business rules pertaining to those fault codes.

Scenario 2 presents a complex example with the need for pattern matching in possibly very large incoming messages. The elapsed time in Figure 12, shows that no generated query required more than five seconds to determine if an interaction existed or not. This shows that DEPICT is time efficient as required by RQ3.

A. Threats to Validity

We qualitatively validate our method and tool DEPICT on two diverse industrial case studies from the Directorate of the Norwegian Customs and Excise (DNCE). A very large empirical study with the application of DEPICT to databases from different companies or contexts is out of the scope of this article. DEPICT is evolving primarily from needs in industrial data-intensive systems and is validated on real industrial case
studies. Nevertheless, the validity of the approach in DEPICT can have several threats as discussed below:

Modelling data interactions with CTM is effective, however, we have not evaluated the scalability of the tool CTE-XL to very large models. We do recommend building many small models representing very specific testing intentions. Large models typically will represent data interactions that show dependencies between several different fields with many different classes. We believe that there is a practical limit to the size and complexity of such dependencies, often linked to the size and complexity of a business rule that is tested. Another aspect of modelling with CTM that has not been validated is its adequacy as a language to model data interactions. We use CTM primarily due to the availability of a well-supported industrial tool CTE-XL. However, it may have limitations as data interaction modelling formalism with growing needs.

DEPICT produces a relational graph from a database schema that needs to be navigated by a human. For most industrial systems the number of tables does not exceed a few hundred (this would make the schema over-complex). However, we do not test the effectiveness of DEPICT for thousands of tables. The time efficiency of DEPICT, for a few millions of records is of the order of a few seconds for all the cases currently modelled by DNCE. We have not validated DEPICT for case studies for SQL joins between many tables with a very large number of records as it was not required by our industrial case study.

DEPICT emerged out of a collaboration with the DNCE and primarily addresses the needs of DNCE. Nevertheless, we have no good reason to believe that DEPICT is not generic for databases from other companies or application domains. The combination of CTM and DEPICT, is adequate to model and verify coverage in any relational database irrespective of underlying database technology. In future, we intend to apply DEPICT to develop the field of model-based data quality using databases at the Norwegian Cancer Registry, StackOverFlow, and Wikipedia.

VI. RELATED WORK

This article addresses two principal areas of work (a) the notion of test coverage in data-intensive systems (b) using high-level models to specify testing intentions. 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 [13]. Test coverage in data intensive systems has been the subject of many studies [21], [27], [34]. 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. [32] measures the coverage of SQL queries without support monitoring coverage. Halfond et al. [11] measure coverage of application-database interactions but do not consider the interactions between database fields. In [14], 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 [35]. 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] [15] [30] [4] [31] [22] [12] [3] [1] [40] [19] [33] [20] [29] that by construction aim to cover a database’s input domain. These approaches are useful when real test databases are non-existent or automatically generated tests that satisfy generic constraints such as cardinality [4] 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 [37] [36] is an effective approach to use behavioral models such as state machines to derive test cases. In [40], 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 [5], 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 [28]. The notion of data interactions proposed in the paper is similar to the idea of data dependencies proposed in [7][8]. 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.

VII. CONCLUSION

In this paper, we present a method and tool 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 propose data interactive coverage as a novel coverage criteria for test databases. We qualitatively validate our approach on industrial test databases extracted from live transaction streams at the DNCE. In future, we would like to leverage DEPICT to promote the general research area of model-based data quality. We would like to use models to represent high-level properties of data and use DEPICT to verify these properties in very large databases (relational, graph XML, JSON).

REFERENCES


