Testing a Data-intensive System with Generated Data Interactions

The Norwegian Customs and Excise Case Study

Sagar Sen[1] Arnaud Gotlieb[1]

Certus Software V&V Center, Simula Research Laboratory, 1325 Lysaker, Norway
{sagar,arnaud}@simula.no,

Abstract. Testing data-intensive systems is paramount to increase our reliance on e-governance services. An incorrectly computed *tax* can have catastrophic consequences in terms of public image. Testers at Norwegian Customs and Excise reveal that faults occur from interactions between database features such as *field values*. Taxation rules, for example, are triggered due to an interaction between 10,000 items, 88 country groups, and 934 tax codes. There are about 12.9 trillion 3-wise interactions. Finding interactions to uncover specific faults is like finding a needle in a haystack. Can we surgically generate a test database for interactions that interest testers? We address this question with a methodology and tool FAKTUM to automatically populate a test database that covers all T-wise interactions for *selected features*. FAKTUM generates a constraint model of interactions in ALLOY and solves it using a divide-and-combine strategy. Our experiments demonstrate scalability of our methodology and we project its industrial applications.

Keywords: database systems, pairwise, T-wise, software product lines, database schema, entity-relationship diagram models, feature models, feature diagram, Alloy, testing

1 Introduction

Data-intensive software are increasingly prominent in driving global processes such as scientific/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 uses the TVINN system to processes about 30,000 declarations a day. TVINN stores validated transactional information such as declarations in a central 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 of semi-structured/structured data in medium/high volume. Testing the data-intensive software systems in the industrial context is the subject of this paper. At Certus Centre, our objective is to identify and provide innovative solutions to thriving industrial problems. Certus is founded

as a consortium with several industry and public administration partners and is primarily funded by the Research Council of Norway. We interact with industry partners in weekly meetings to understand and help solve their problems using state-of-the-art research findings and tools. We go by the motto-*Industry is our lab!*, owing to the fact that our scientific problems emerge from industry. This paper narrates the story of our collaboration with the Norwegian Customs and Excise department who bring to us the problem of testing the TVINN system.

The TVINN system at the Norwegian Customs and Excise department has been in operation since 1988. It is a massive database application that processes declarations sent as standard EDIFACT messages from customs officers and companies concerned with import/export in Norway. A sophisticated script called EMIL, written in the legacy language Sysdul, processes about 30,000 declarations/day to notify customs officers about the correctness of the declarations. The correctness is verified based on a large set of well-formedness rules and customs laws and regulations. For instance, if the net weight is less than half the gross weight the script identifies a problem and sends back a declaration for further evaluation by a company/customs officer. The customs laws change periodically with new governmental policies. For example, there has been a steep hike in taxes for import of cheese from Europe to protect Norwegian farmers. Customs laws also can be change for a short period of time by Customs officers who want to enforce a law on a series of suspicious trucks. Therefore, the dynamic change of rules in the TVINN system makes testing it constant effort. Moreover, a recent major development in the TVINN system is its migration from Sysdul to a system developed in Java. Will the new system function equivalent to the one currently under operation? Will the system regress? This is the fear that a public service such has to face. Current practice in testing involves domain experts manually creating test databases that aim to reveal faults and differences between different versions of TVINN. Talking to testing experts we understand that manual test creation is tedious and most importantly its hard to know where the tester's journey ends. There is a lack of the notion of *test coverage*. In this paper, we outline Certus center's efforts to address this particular challenge.

We present a methodology to automatically generate test data that cover all valid T-wise interactions between database features. These features correspond to *selected field values* in the specification of a database's *input domain* which is the *database schema*. We use the *feature modelling formalism* to help a test engineer select variation points in a database schema as a feature model. It also allows the tester to specify forbidden feature combinations and feature interdependencies. Using a feature model as input we apply our previously developed approach in [1] to generate a set of *configurations* that cover all valid T-wise interactions between database fields of choice. This configuration generation module is scalable to large feature models and satisfies all feature interdependencies in the configuration. The approach is based on transforming a feature model and T-wise combinations to a set of constraint systems in Alloy. These systems are solved concurrently and based on a *divide-and-combine* strategy. The solutions

are concurrently generated as sets of configurations that are combined into a final set. We transform these configurations to insertion queries to populate a test database. However, certain fields are still placeholders for values in their appropriate domains. The fields that have little significance to a tester are updated using generated SQL update queries. The update queries either update values to maintain *data integrity* (domain and referential integrity) or random values for fields with less significance to the tester. The result is a complete set of queries to populate a test database covering all valid T-wise interactions between database features of interest. We implemented the methodology in a prototype tool called FAKTUM (fact in Norwegian).

We performed experiments to validate our methodology. We used a *common database schema* developed in collaboration with the Norwegian Customs and Excise as the input domain and specify a feature model to select data features concerning imports from Brazil, India, China, and the USA. We generated a test database using these inputs that covers all 2-wise/pairwise interactions between a total of 75 features such as currencies, country codes, country groups, tax fee codes, and declaration categories to name a few. We generated about 935 configurations covering all valid pairwise interactions (of 10804 all possible pairwise interaction) between the chosen features. These 935 configurations were transformed to a set of Structured Query Language (SQL) queries to populate a test database. The time complexity to detect the validity of a single pair (/tuple) is $\mathcal{O}(1)$. The average time to detect validity is about 12 ms. We generated 187 sets of configurations covering divided subsets of pairs. The generation of configurations is time and scope bounded. Every call to Alloy's SAT solver is bounded to an average of 400 ms and 6803 calls to the solver were necessary to obtain the final set. The large number of calls to Alloy's SAT solver is a trade-off to address scalability. We combined a total of 935 configurations from 187 sets that were solved concurrently in 48 minutes.

We summarize our contributions as follows:

Contribution 1: We propose a scalable methodology and a prototype tool FAKTUM to automatically generate databases that cover all T-wise interactions between data fields of testing interest.
Contribution 2: We demonstrate through experiments on an industrial case study that our methodology is scalable and can be used surgically generate tests.

The paper is organized as follows. In Section 2, we present a detailed overview of the case study at the Norwegian Customs and Excise and the problems associated with the complexity of the testing task. In Section 3, we present a methodology based on automatic generation of test databases satisfying T-wise interactions to address the testing problem at the Norwegian Customs and Excise. We present results from a concrete experiment in Section 4. Section 5 discusses of related work, while we conclude with a summary of our experience in Section 6.

2 Industrial Case Study: Norwegian Customs and Excise

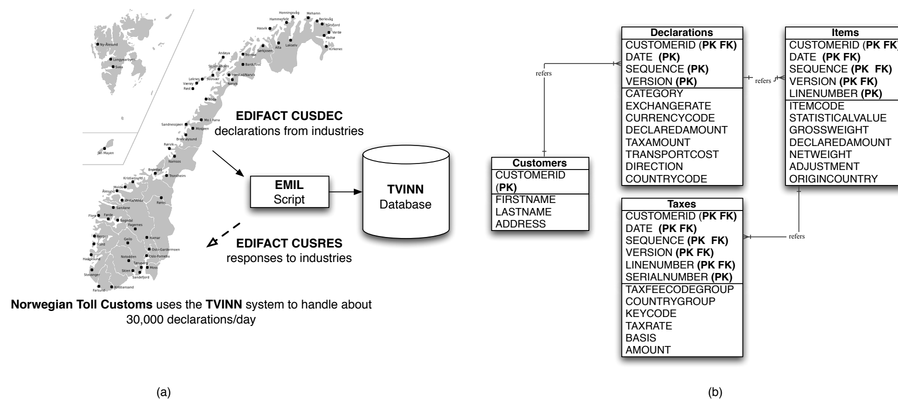

Fig. 1. (a) Norwegian Toll Customs Industrial Case Study (b) Common Database Schema at Norwegian Toll Customs in Crow-Foot Notation

We describe our industrial case study from the Norwegian Customs and Excise as illustrated in Figure 1(a). The system under study is the *Tollvesenets startside for internettfortolling* (TVINN). An overview of TVINN process flow is presented in Figure 1(a) and official description is available at https://fortolling.toll.no/Tvinn-Internett. Customs officers and industries associated with import/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 script 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) taxes are computed based on about 900 tax code groups (d) an list of more than 10,000 items. A declaration can be categorized into 6 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 (approx. every 6 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 time-bounded rules are ephemeral and exist for a short period of time. For instance, a customs officer decides to thoroughly check 20 trucks coming from a nation X in civil war. He/she will possibly create a rule to check all trucks from X for the next 3 hours. This rule is called *mask control* will

disappear after 3 hours. These rules can change on an everyday basis without anticipation making TVINN a highly dynamic system.

TVINN is a complex and dynamic database application that processes about 30,000 declarations per day. Testing TVINN is a challenge since it evolves rapidly. Moreover, in 2012-2013 TVINN will undergo a migration from its native implementation Sysdul to Java. Will the new implementation in Java regress with respect to the old one? This is the question that haunts the Norwegian Customs and Excise at present and in the years to come.

Testing TVINN has been intuitively achieved by a small testing staff executing a subset of a large number of readily available records of *real-world declarations*. However, using these *real-world declarations or records* present four important problems:

No Coverage Guarantee: Records obtained from real-world transactions such as customs 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.

Very Large Set of Test Records: Accumulating information from real-world 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 records that precisely captures testing intentions. 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, its often impossible for them to outsource their testing efforts to external agencies who would use real-world records.

Constantly Changing Rules: Records for testing often have a lifetime and need to be discarded. For instance, in the Norwegian Customs and Excise system changes when sanctions are imposed on countries or significant changes happen in currency exchange rates. Legacy transaction records may not be used anymore to test the evolved system.

Our objective at Certus Software V&V center, Simula Research Laboratory, is to present a solution to address these problems.

3 Methodology

In this section, we present a methodology to generate test databases that cover all T-wise interactions between data fields of interest. The overview of our methodology is shown in Figure 3 (a). In Section 3.1, we present foundational notions that will be used to describe our methodology. We present the different steps in our methodology in Section 3.2. In Section 3.3, we describe the implementation of the methodology in our tool FAKTUM.

3.1 Foundations

Database Schema The first input artifact in our methodology is a *database schema*. It specifies the input domain of a database. We briefly describe the well-known concept of a database schema. More information on them can be found in a standard database textbook such as in [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 contain one or more fields that are referred to as *primary keys*. This means that each record is identified by its primary key. Table may refer to primary keys in 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 a database schema. Databases can be queried using Structured Query Language (SQL) queries. These queries are both used to create and populate a database and query it for information presented as a table or a view. In this paper, we generate several hundred SQL INSERT and UPDATE queries to populate a test database (see Section 3.2 and Section 4 for more information on generated queries).

As a running example, we present a schema developed along with our industry partner, the Norwegian Customs and Excise, in Figure 1(b). The database schema 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 is 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.

Feature Model of Database Variability Populating the database schema requires selection amongst a set of choices for its field values. The second input artifact to our methodology is a model of variation points or choices in the database schema. We use the feature modelling formalism to specify the variability in a database schema. The feature modelling formalism is described in detail in [3]. Typically, a feature model is used to specify the different features

in a *software product line* and their inter-dependencies. Inter-dependencies are constraints on the choice of features. Some features are mandatory, some are optional, some features require other features while some features are mutually exclusive (XOR) with respect to other features. Features can be *abstract* or *concrete*. Abstract feature help in classification and hierarchy while concrete features go into a final software configuration. A configuration of a feature model is a finite set of concrete features that satisfy feature model constraints. The notion of feature models is very popular and a general formalism to specify variability in software product lines and software artifacts in general. We use the feature modelling formalism in this paper to specify variability in a database schema.

In Figure 2, we present the feature model of database variability in the Norwegian Customs and Excise schema. The mandatory root feature specifies the database identifier which is TollCustoms. All child features of root in the second level specify identifiers for the different tables available in the database. The third level contains features for database fields in each of the tables. In the fourth level of model the features specify the different possible values for each field. The database, table, and field features in the first, second, and third level respectively are mandatory but abstract features. The values for fields are mutually exclusive which means only one value can be associated to a field. For instance, Declarations is table feature that has a Category feature. There are six different declaration categories. Only one of the values such as FU can be associated to the category field. We present the field values in a concise manner in Figure 2. We show the first possible variation in a field value while we only show the number of other possible possible values due to space limitations. A selection of field value features for all fields across all tables is what we call a *record configuration*. A record configuration specifies the exact values that will go into each field for a record in each and every table of the database. We use the term configuration interchangeably with record configuration. Multiple record configurations represents multiple records that go into the different tables of the database. For instance, $[FU, USD, BR, Import, ...]$ is part of a record configuration that shows values that will go in as a record in the Declarations table.

Combinatorial Interaction Testing *Combinatorial interaction testing* (CIT) is a recognized software testing technique introduced by Cohen [4], that tests all interactions between features, parameter values or in our case database field vales. An **interaction** can be seen as a **tuple of software features**. A widely cited NIST study of the fault databases of several real-life systems reports that all the faults in these databases are caused by no more than six factors [5]. If test features are modelled effectively, a T-wise testing can expose all the faults that involve no more than T features. However, pairwise or 2-way testing has been shown to be both time efficient and effective for most real case studies http://www.pairwise.org/results.asp. This motivated us to focus on CIT of all pairwise interactions of database field values of interest (see Section 4). T-wise testing requires that every T-wise interaction of between all features/database

field values is present at least once in a set of record configurations. Generating record configurations that cover all T-wise interactions is a challenging task for very large feature models. There is a combinatorial explosion in the number of possible interactions with the increase in the number of features and the value of T. For instance, 10,000 items codes, 88 country groups, and 934 tax fee codes give rise to 192.5 million pairwise/2-wise and 12.9 trillion 3-wise interactions. In [1], we present a scalable approach and tool to generate a set of configurations that cover all T-wise interactions between features. We use the approach in [1] to generate a set of *record configurations* that cover all T-wise interactions between database field values. Although, our approach makes the problem tractable it still depends on the computing resources (running in parallel) for a very large number of interactions. Therefore, in our experiments we go one step further and surgically select only a relevant subset of field values.

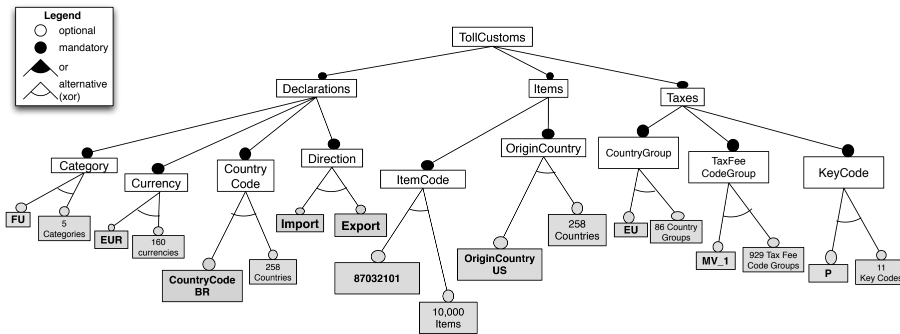

Fig. 2. Feature Model of Database Variability

3.2 Methodology Description

We describe the methodology in four phases where each phase subsumes several sub-steps. An illustrative overview of the methodology is given in Figure 3 (a).
Phase 1. Tester interaction: A tester provides three inputs (a) A database schema specification as described in Section 3.1. (b) Variability in a database as a feature model. The variations in database field values for TollCustoms is shown in Figure 2 (c) The value of T which represents the strength of interactions needed to be tested. For instance, when $T = 2$ we intend to test all 2-wise or pairwise interactions between database field values. If we consider all pairwise interactions between two features declaration category FU and currency code USD we have 2^2 different interactions as shown in Figure 3(b). Similarly, we have 2^3 possible 3-wise interactions between a set of 3 features. In general, with n features we have $2^T \times \binom{n}{T}$ possible interactions minus those interactions that are forbidden by feature model inter-dependencies.

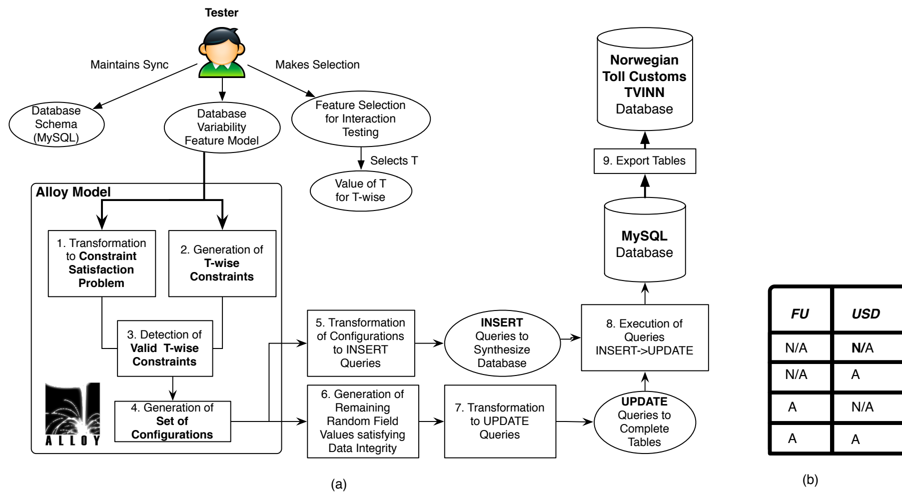

Fig. 3. (a) Methodology (b) Pairwise Interactions Between Two Field Values

Phase 2. Generation of database configurations covering all T-wise field interactions: A *record configuration* specifies the field values for each record (across multiple tables) in a database. Therefore, we first generate a set of configurations that cover all T-wise interactions between concrete features (in grey) specified in a feature model such as in Figure 2. This is achieved via steps enumerated 1-4 in Figure 3(a). These steps are derived from our previous work in [1]. We briefly describe the approach from [1] here. In Step 1, we automatically transform a feature model to a constraint satisfaction problem **A** in the formal language Alloy [6][7]. In Step 2, we generate Alloy predicates that encode *tuples of features* representing T-wise interactions between *concrete field value features* (shown in grey in Figure 2). We insert these predicates into the Alloy model **A**. In Step 3, we detect all T-wise Alloy predicates consistent with **A** and reject the others which are not accepted by the feature model specification. In Step 4, we use a *divide-and-combine* strategy to generate sets of configurations that satisfy interaction tuples divided in subsets. We combine the sets to obtain a set of configurations that cover all T-wise interactions between features.

Phase 3. Transformation to SQL queries: The record configurations generated in the previous step need to be transformed into SQL queries to populate a test database. There are three steps 5-7 from Figure 3(a) involved in this phase. In Step 5, we generate SQL INSERT queries to populate a database with record configurations that cover T-wise interactions. The field values in a record configuration are transformed to an INSERT query in a straightforward manner. For instance, an INSERT query for the Declarations table is shown in Listing 1.1.

```
INSERT INTO Declarations(Category, Direction, CountryCode, CurrencyCode)
VALUES ('FO','I','US','CNY');
```

Listing 1.1. Example INSERT query for a Declarations Record

However, we notice that the INSERT query only contains values for a subset of all fields in record. The remaining fields have a NULL value. Therefore, in Step 6, we generate UPDATE SQL queries to fill in values for the remaining fields. The values of the remaining fields can be pseudo-randomly generated in their respective domains while satisfying *data integrity* constraints. For instance, a value for CustomerID in the declaration table must be an numeric string with at least 8 digits (domain integrity) and every customer id must be present in the CUSTOMERS table (referential integrity). In our methodology, coverage of T-wise interactions has priority. Therefore, we generate unique random values for all remaining fields. All foreign key field values are identical to their primary key values to ensure referential integrity. In Listing 1.2, we illustrate an UPDATE SQL query to complete the partial record created using INSERT query in Listing 1.1.

```
UPDATE INTO Declarations(CustomerID, Date, Sequence, Version, Amount, FeeAmount,
                          TransportCost, ExchangeRate)
VALUES ('2002542616','1965-3-29','1','1','2982.490245', '1343.471627',
                          '79.0749637','112.7416998');
```

Listing 1.2. UPDATE query to Complete a Declarations Record

Phase 4. Population of Test Database: The last phase of our methodology is a straightforward population of a MySQL test database. The database is populated using INSERT queries followed by UPDATE queries (as shown in Step 8, Figure 3 (a)) generated in Phase 3 of our methodology. The test database generated in for the common database schema can be exported to more sophisticated industrial data-intensive systems in Step 9. This separation of database specifications allow us to work on a testing-specific subset of the entire database at Norwegian Customs and Excise. The full TVINN database contains several entities and fields that do not interest testers. The collaborative effort with the Norwegian Customs and Excise department helped us extract a testing specific subset with *bi-directional exportability* of records.

3.3 Implementation in Faktum

We implement the database synthesis tool FAKTUM for T-wise interaction testing in Java. The implementation is standalone and uses libraries for constraint solving such as ALLOY. A prototypical implementation of the tool to generate *configurations* is available online https://sites.google.com/a/simula.no/dbtwise/.

4 Experiments

We perform an experiment to synthesize a test database for the Norwegian Customs and Excise case study and discuss the database synthesis results, scalability, and threats to validity of our approach.

4.1 Experimental Setup

We develop a test scenario to automatically synthesize a database that will test rules applying to Norwegian imports/exports from/to Brazil, India, China and USA. The feature model of database variability is specified in Figure 4. This is a subset of the full feature model shown earlier in Figure 2. This feature model specifies the variations of *fields of interest* in the database schema shown in Figure 1(b). This selection of features surgically pin-points a specific testing zone in the vast input domain. We use the feature model in Figure 4 as input to generate a set of configurations that cover all 2-wise/pairwise interactions between field values. The total number of interactions is the number of interactions between 37 concrete features minus the invalid interactions due to the XOR relation between field values in the same field. There are $4 \times \binom{37}{2} - \binom{5}{2} - \binom{4}{2} - \binom{4}{2} - 2 \times \binom{2}{2} - \binom{4}{2} - \binom{5}{2} - \binom{4}{2} - \binom{9}{2} = 2582$ pairwise interactions that need to be covered by a set of record configurations.

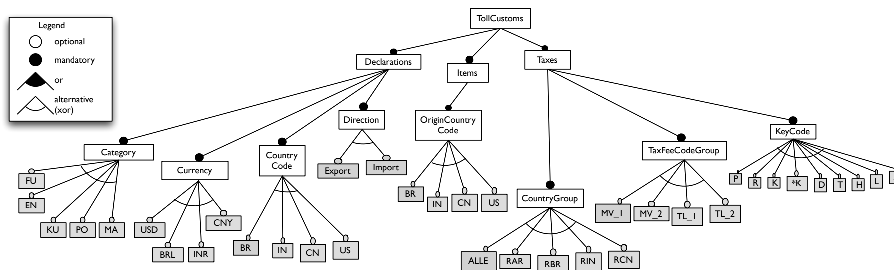

Fig. 4. Feature Model with Features Selected for Interaction Testing

4.2 Results of Discussion

We generate **935 record configurations** covering all valid 2582 pairwise interactions of features in Figure 4. In the worst case, a naive non-optimal set of configurations will have the size 2582 with one interaction covered per configuration. These record configurations are transformed to SQL queries to populate a test database. The set of SQL queries are available in https://sites.google.com/a/simula.no/dbtwise/. In this paper, we focus on understanding the scalability of our approach for the industrial case study.

An important step in our methodology verifies if an interaction between database field values is indeed consistent with respect to its feature model. Every interaction is a tuple of field values that must be present in a configuration. However, not all tuples are valid with respect to the feature model in Figure 4. For instance, a tuple cannot contain two different categories for a declaration since all categories are mutually exclusive (XOR). Therefore, these *forbidden*

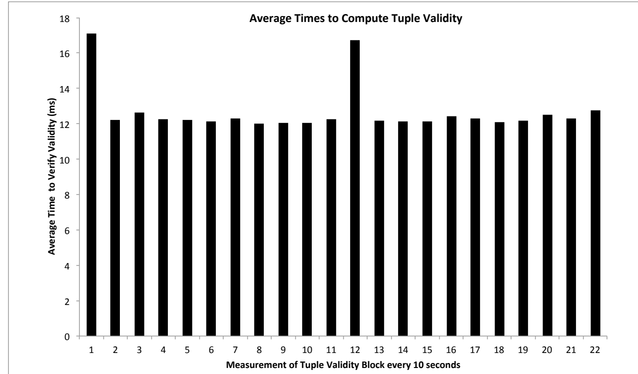

Fig. 5. Checking Tuple Validity has $\mathcal{O}(1)$ Time Complexity

tuples must be weeded out. Each tuple is transformed to an ALLOY predicate and concurrently solved with the ALLOY model the complete feature model. If the predicate conflicts with the ALLOY model and its facts then its discarded. Checking tuple validity has a time complexity of $\mathcal{O}(1)$ for a feature model of finite size. This is experimentally validated in Figure 5. We measure the time spent in the block of code for checking tuple validity. We observe that the average time in this block is about 12 ms when measured every 10 seconds. We use the non-intrusive perf4J library to perform the measurements.

Another important step is the generation of a set of configurations satisfying all valid tuples of feature interactions. We use the divide-and-combine approach to concurrently generate configurations satisfying all interaction tuples divided in exclusive subsets. Our methodology creates 187 concurrent sets of configurations satisfying the 2582 interaction tuples. The total number of configurations in the 187 sets is 935. We measure the number of **calls to the constraint solver** as shown in Figure 6. Creating the 935 configuration required about 6804 calls to ALLOY's SAT solver. However, the average time spent in the solver was within about 450 ms due to the finite scope of the solver. The large number of calls to a constraint solver is a trade-off to achieve high scalability. Theoretically, we can imagine just one call to a constraint solver attempting to generate a set of configurations satisfying all 2582 interactions. However, in practice this is intractable and our divide-and-combine strategy address this exact issue.

4.3 Threats to Validity

Our experiments do not consider a certain number of factors that could affect our generalized outcomes on scalability. The scalability of our methodology may most likely be affected by the increase in the number and complexity of constraints in the feature model. However, we suggest surgically creating test models covering interactions between only relevant features for a tester. This will most often keep the satisfaction problem tractable. Another, factor we did not consider

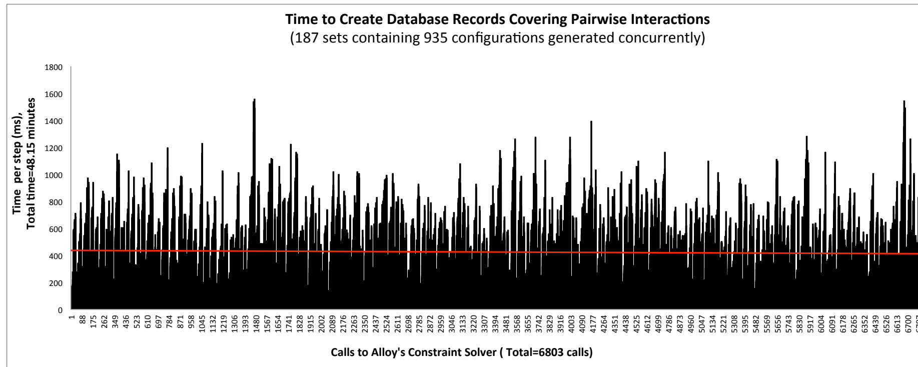

Fig. 6. Concurrent Configuration Generation

is presence of numerical constraints between database fields that are generated randomly. For instance, the value of computed tax for an item is a function of the statistical value of an item. In this paper, these database fields were associated to random values without conforming to numerical constraints since in the general intention for testing was the interaction and not he numerical correctness of fields storing real numbered values. An extension to our tool FAKTUM will be to ensure that numerical constraints are satisfied between real-valued fields.

5 Related Work

The related work for this paper falls into two areas: (a) generation of test configurations covering T-wise interactions in a feature model (b) generation of synthetic data for databases. We will review the some relevant essential contributions in these areas.

Researchers have proposed several approaches for generating test configurations satisfying T-wise and notably the 2-wise/pairwise coverage criteria. The pioneering work in this area is the AETG approach [8] and its implementation to address highly-configurable software systems [9]. Based on a greedy algorithm, AETG generates N-wise covering arrays for a set of parameters and converts these arrays into a set of test cases. However, it cannot deal with constraints among the parameters, thus limiting its adoption to feature models with constraints. Oster et. al. [10] uses a greedy and ad-hoc algorithm based on the maximum number of valid pairs within each configuration while satisfying constraints. Recently, Johansen [11] introduced the SPLCATool (Software Product Line Covering Array Tool) to generate test configurations from feature models. Similar to the above mentioned approaches, the tool uses greedy algorithm to enforce all pairs in a set of configurations. The tool is quite efficient and has been used on large feature models. The tool PACOGEN developed by Hervieu et. al. [12] goes a step further and generates a *minimal set* of configurations that covers all pairwise using a time-aware constraint solving procedure. However, for the

problem of generating database records covering all T-wise interactions it was necessary to populate a test database with the generated configurations. Therefore, we used our previously developed approach described in Perrouin et.al. [1] to generate a set of configurations covering T-wise and satisfying feature model constraints. The approach in [1] is scalable as it based on a divide-and-combine strategy. According to our knowledge, this is the first time an interaction testing tool is used to populate a test database for data-intensive systems.

Important work to generate synthetic databases include [13], [14], [15], [16]. Database generation tools such as in [14], [17], [13] allow users to specify the data distributions over attributes and intra-attribute correlations. Houkjaer et al. [14] use a graph model containing primary-foreign keys. This model is used to guide the data generation process. In all these works, the question of generating general T-wise interactions-covering data is not addressed and considering constraints among the fields is irrelevant. In that respect, the methodology proposed in this paper innovates also w.r.t. to database generation tools.

6 Conclusion

In this paper, we address the problem of testing TVINN, a data-intensive system in the Norwegian Customs and Excise department. TVINN processes all custom declaration coming in and out of Norway. We introduce a methodology implemented in a tool called FAKTUM, to populate a test database with configurations that cover all T-wise interactions between selected custom declaration database fields. In an experiment, we develop a variability model (feature model) and a database schema both developed in collaboration with the Norwegian Customs and Excise department. We use our tool to automatically generate 935 configurations covering all valid 2582 2-wise/pairwise interactions of features in less than 48 minutes. The scalability of our approach and the possibility to generate test data covering all T-wise interactions has given way to *food for thought* in our academia-industry partnership. For instance, we realize that random elements, in the test database, have a low degree of comprehensibility for testers at TVINN. Hence, we propose the use of FAKTUM for partial population of test databases. The remaining elements are manually completed by testers at Norwegian Customs and Excise. The partial test data gives testers the confidence of covering all valid T-wise interactions. Our future work with our partner will involve a large-scale empirical evaluation of test generation while carefully taking into account the clauses of non-disclosure agreements.

7 Acknowledgement

We thank the Norwegian Customs and Excise department for their trustful interactions with us. In particular, we would like to thank Atle Sander, Astrid Grime and Katrine Langset for their valuable inputs. We thank the Research Council of Norway for their generous support ; it would not have been impossible

to setup such a close industry-academia collaboration for high impact software engineering research without it.

References

1. Perrouin, G., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Automated and scalable t-wise test case generation strategies for software product lines. In: International Conference on Software Testing (ICST'10), Paris, France (2010)
2. Date, C.J.: An Introduction to Database Systems. 8. edn. Pearson Addison-Wesley, Boston, MA (2004)
3. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC. (2005) 7–20
4. Cohen, D., Dalal, S., Fredman, M., Patton, G.: The aetg system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering **23**(7) (1997) 437444
5. Kuhn, D.R., Reilly, M.J.: An investigation of the applicability of design of experiments to software testing (2002)
6. Software Design Group, M.: Alloy community. http://alloy.mit.edu (2008)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (March 2006)
8. Cohen, D.M., Society, I.C., Dalal, S.R., Fredman, M.L., Patton, G.C.: The aetg system: an approach to testing based on combinatorial design. IEEE Trans. SW. Eng. **23** (1997) 437–444
9. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering **34**(5) (2008) 633–650
10. Oster, S., Markert, F., Ritter, P.: Automated incremental pairwise testing of software product lines. In: Software Product Line Conference (SPLC'10). (2010)
11. Johansen, M.F., Haugen, O., Fleurey, F.: Properties of realistic feature models make combinatorial testing of product lines feasible. In: Conference on Model Driven Engineering Languages and Systems (MODELS'11). (2011) 638–652
12. Hervieu, A., Baudry, B., Gotlieb, A.: Pacogen: Automatic generation of pairwise test configurations from feature models. In: Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium. (2011) 120–129
13. Bruno, N., Chaudhuri, S.: Flexible database generators. VLDB (2005) 10971107
14. K. Houkjaer, K.T., Wind, R.: Simple and realistic data generation. VLDB (2006) 12431246
15. E. Lo, N.C., Hon, W.K.: Generating databases for query workloads. VLDB (2010) 848 - 859
16. Mannila, H., Raiha, K.J.: Automatic generation of test data for relational queries. . J. Comp. Syst. Sci **38**(2) (1989) 240258
17. IBM: IBM DB2 test data generator. http://www.ibm.com/developerworks/data/library/techarticle/dm-0706salkosuo/index.html