UNIVERSITY OF GOTHENBURG

Model Based Testing of Data Constraints

Testing the Business Logic of a Mnesia Application with Quviq QuickCheck

Nicolae Paladi

Bachelor Thesis in Software Engineering

Report No. 2009:004
ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2009

Abstract

Correct implementation of data constraints, such as referential integrity constraints and business rules is an essential precondition for data consistency. Though most modern commercial DBMSs support data constraints, the latter are often implemented in the business logic of the applications. This is especially true for non relational DBMS like Mnesia, which do not provide constraints enforcement mechanisms. This case study examines a database application which uses Mnesia as data storage in order to determine, express and test data constraints with Quviq QuickCheck, adopting a model-based testing approach. Some of the important stages of the study described in the article are: reverse engineering of the database, analysis of the obtained database structure diagrams and extraction of data constraint, validation of constraints, formulating the test specifications and finally running the generated test suits. As a result of running the test suits randomly generated by QuickCheck, we have detected several violations of the identified and validated business rules. We have found that the applied methodology is suitable for applications using non relational, unnormalized databases. It is important to note the methodology applied within the case study is not bound to a specific application or DBMS, and can be applied to other database applications.

1. Introduction

Referential integrity is the database-related practice of ensuring that implied relationships between tables are enforced. Most modern Database Management Systems (DBMSs), especially *relational* DBMSs have built in mechanisms for defining and ensuring basic data constraints [24]. However, in practice far from all constraints are defined in the database management system itself, but many are rather encoded in the application using the data. For example, an application supporting an internet shop would impose a relation between a customer willing to make a purchase and the credit balance of that customer. In a purely relational database, one could hard code that the credit must at least be the purchase amount, but this is hardly ever done, since this constraint is based on a business strategy that may well change or is different for different customers.

Many database applications have a layered architecture, part of the data constraints are hard coded as relational constraints in the DBMS, other constraints are implemented in the business logic of the application. There are several reasons for implementing constraints in the business logic rather than in the DBMS. For example, the above mentioned situation in which one wants to get flexibility in the relationship; either in the future or for special subset of the customers. Other reasons may be purely social, such as lack of developer time or required expertise and insight, or strictly technical. An excellent example of the latter case is Mnesia [26], a distributed DBMS, appropriate for telecommunications applications and other Erlang [2] applications which requires continuous operation and exhibit soft real-time properties. According to Mattsson et al, [26], Mnesia employs an *extended relational model*, which results in the ability to store arbitrary Erlang terms in the attribute fields. However, Mnesia is not a relational database and does not have any mechanisms for ensuring database constraints other than ensuring them in the business logic of the application.

Problem: ensuring the constraints

When data constraints cannot be ensured by a DBMS alone, then those constraints are much less visible in the software design. As a consequence, constraints are often implicitly defined and certain parts of an application may accidentally violate a constraint. Constraint violation may result the database to be in an inconsistent state and software assuming certain properties of the data may crash. In addition, violation of constraints may impact the business, since it may be possible to perform actions that the business disallows.

When software needs to be reliable, the constraints implemented in the business logic need to be satisfied and therefore it should be tested that they cannot be violated by executing the application.

Defining and enforcing database constraints within relational (and to a lesser extent object oriented) databases has long been in the focus of both academia and industry. The SQL implementation [17] of database constraints is currently supported by most relational DBMSs. A simple example is that the SQL standard for ensuring referential integrity (which is a typical example of a database constraint) is supported by DBMS like MySQL, Microsoft SQL Server, DB2, Oracle and even MS Access (through its graphical relational tool) [9]. However, this does only solve simple relations and not the more dynamic relations captured by the business rules.

There has been little research done on the topic of ensuring business rule constraints, especially when we consider databases which do not use SQL. Furthermore, previous research assume the database to be in at least the 3rd normal form [22] or higher, and do not consider the case of unnormalized databases [14], [12], [15]. Nevertheless, non-relational unnormalized databases are rapidly becoming more popular.

The case: a large Mnesia application

Recently, Castro and Arts [4] have developed a method for testing business logic constraints with Quviq QuickCheck. They used their method to verify business logic constraints in an application on top of a normalized relational database. In this paper we show that the method is also applicable to unnormalized, non-relational databases and that we are able to identify business logic violations in existing applications.

Kreditor AB is a Stockholm-based financial organization which has developed for its operations a database application implemented in Erlang using Mnesia for data storage. The application, further referred to as the *Kred application*,

uses an unnormalized database supported by a non relational database system. ***In this paper we show that the method of Arts and Castro is applicable to Kreditor's database application. We show that we can identify violations of the constraints. Therefore, it will help improve the existing solutions for testing data constraints for non relational database systems, and minimize the occurrence of situations when invalid data input can lead to data corruption.***

As part of this study, we have reverse engineered the Kred application to create its database schema and the corresponding entity-relationship (ER) diagram. Besides, we have identified a number of data constraints that are implemented in the business logic of the application.

The presented approach is generally applicable to non-relational databases, but in particular to Erlang applications build upon Mnesia.

2. Related Research

We base our case study on the method developed by Castro and Arts [4], which is a general methodology for testing data consistency of database applications. In this approach, the system under test is modeled as a state machine, the state of which is examined after consecutive calls to database interface functions. In the method, the focus lies on keeping the state as simple as possible and not making the state a copy of the database; only data generated by the interface functions (such as unique keys, etc), should be stored in the state, the rest is assumed to be correctly stored. The state machine model is tested against the real application with QuickCheck, (cf. [5]). The novelty of the method of Castro and Arts is that business rules are formulated as data invariants and are checked after each test.

The method is applied to a normalized, relational database and invariants are described and executed as SQL queries.

2.1 Other approaches

Chan and Cheung support the idea that current "traditional" approaches in software testing cannot reveal many of the software errors which can lead to database corruption. Therefore, they suggest the idea of extending the white box testing approach with the inclusion of SQL statements that are embedded into the database application. In order to do that, they suggest to convert the SQL statements to the general programming language in which the application is implemented and include them into the white box testing framework [10].

In addition, Chan *et al* propose to integrate SQL statements and the conceptual data models of an application for fault-based testing. In their paper, they propose a set of mutation operators based on the standard types of constraints used in the enhanced entity-relationship model. The operators are semantic in nature and guide the construction of affected attributes and join conditions of mutated SQL statements [11].

Chays *et al* have developed a framework for testing relational database applications called AGENDA. AGENDA has a strong reliance on the relational model and SQL and its use has not been described for non-relational databases [12].

Dietrich and Paschke describe a test-driven approach to the development and validation of business rules [16]. They propose a way to develop JUnit test cases based on formal rules, however they propose a manual implementation of the test cases.

As a complement to the above method, Kuliamin's description of the UniTestK test development technology [25] contains some practical advice on using models to test large complex systems. In particular, the author describes the use of well known software engineering concepts such as modularization, abstraction and separation of concerns in order to manage the stages of determining the interface functions, development of the model, and finally the development of the test scenario.

3. Research Approach

The project has been carried out with an emphasis on quantitative post positivist approach, focused on a combination of qualitative in-depth analysis of the database application under examination, and empirical observation of the results of an extensive set of randomly generates test instances.

In the light of Boudreau's claim that "Field experiments involve the experimental manipulation of one or more variables within a naturally occurring system and subsequent measurement of the impact of the manipulation on one or more dependent variables" [8], this study heavily relies on field experiments which will focus on studying the change of the variables in the Kred application as a response to certain alterations of the database. Furthermore, during the study we have not only observed and measured the occurrence of changes, but also compared them with the *expected* alterations. Based on the outcome of the latter comparison, we have been able to draw conclusions on whether the business logic of the system conforms to the requirement of maintaining the data in a consistent state. Other methods used include interviews [23], data analysis, and heuristic estimations of the functionality limits of the system under examination for test design purposes [27].

In the course of the project we had to answer several questions concerned with database representation, identification of database constraints, as well as their codification. This section will describe the tools used, as well as the steps taken to conduct this study.

3.1 Limitations of the study

This study focused on database applications implemented in Erlang and which use Mnesia as data storage. Despite the fact that both Erlang and Mnesia have highly concurrent and

distributed properties, such aspects have not been taken into consideration in the current study.

3.2 Tools

3.2.1 Test generation tools

In order to fully leverage the power of the formal verification approach adopted for testing the business logic, we choose QuickCheck to generate and execute the tests. Quviq QuickCheck is a specification based testing tool [5] which tests the software with randomly generated test cases, which follow a formal specification expressed in Erlang. QuickCheck has several libraries for expressing higher level system specifications like the state machine library that we used.

There are several other test generation tools available, which are listed below [1]:

- TVEDA, a tool developed by France Telecom CNET [28], which generates tests against formal specifications written in TTCN, which is an ISO test suit notation standard [31], [34]. This tool is used by France Telecom, mainly for testing telecommunication protocols.
- TorX is an *on-the-fly* testing tool. i.e. which offers support for test generation and test execution in an integrated manner. It generates tests against specifications expressed in PROMELA and LOTOS [30].
- Blom and Jonsson describe a case study of automatic test generation for a telecom application implemented in Erlang. In their detailed paper, the authors also describe the test generation algorithm, as well as a formal specification language, Erlang-EFSM [7]. However, it is not fully developed and has not moved further from the concept state described in the article.

QuickCheck's support of Erlang and library for state machines, together with a larger number of previous case studies, has made it the preferred tool for our research project. However, it is important to note that the method followed in our case study is generalizable, and it is not strictly bound to either QuickCheck or the Kred application.

3.2.2 Structure visualization

As a consequence of the compelling lack of suitable database reverse engineering tools as well as of database structure visualization tools that could be used for Mnesia, Dia has been used to visualize the database structure, both the ER diagram and the Database schema. Dia is a lightweight open source tool that has been chosen particularly for its relatively extensive capabilities [35]. While Dia may not be a specialized database visualization tool, its capabilities allow to plot the structure of databases as complex as the database used by the Kred application.

[1] Far from being a complete list, this is an example selection of test generation and execution tools

3.3 Examination of the database structure

One of the first steps in our work has been the manual examination of the database structure. Reverse engineering of relational databases has been in the focus of research, and several approaches are available.

To mention a few, Premerlany and Blaha offer a generic approach to reverse-engineering legacy relational databases [29]. In their paper the authors describe a manual process of analysis, deconstruction and visualization of the database model, which consists of seven steps. Premerlany and Blaha support the idea that reverse engineering of legacy databases should be carried out in a flexible, interactive approach. The authors also claim that an approach based on rigid, batch-oriented compilers will most likely fail [29].

Similarly, Andersson describes the process of Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering [1]. In his approach, Andresson also use an ER model extended with multi-valued and complex data, as well as multi-instantiation. This makes the latter approach suitable for reverse engineering of the database under assessment.

The method described by Premerlany and Blaha has numerous similarities with the current study, and in particular the focus on large unnormalized databases, the consideration of the lack of enforcement for foreign keys, as well as consideration of the "optimized or flawed schemas which are often found in practice" , [19]. Furthermore, this approach is suggested by the authors as suitable for large legacy databases with little or no semantic advice available [29]. However, this method is not entirely applicable, mainly due to its focus on *relational* database systems, as well as due to the large effort required to reverse engineer the database structure, especially since reverse engineering of Mnesia databases is *not* a central issue for this project.

Following the above idea, reverse engineering of the Kred database has been performed in an iterative process consisting of the steps described below.

- *Determine Candidate Keys*, a step focusing on identifying the primary keys of the tables. In case of Mnesia, this is facilitated by the peculiarities of record definition, where the *first* element of the record serves as a key to the record.
- *Determining Foreign-key Groups* by observation of the tables' elements, search for synonyms, homonyms, and fields with the same name.
- *Discovering associations* by revealing additional links of all types between the tables, with the help of code comments and other semantic information. However as Premerlany and Blaha state, "one should be careful at this stage, since reverse engineering produces hypothesis, which must nevertheless be validated with the help of semantic understanding".

- *Performing the transformation* by transferring of the discovered information, based on decisions on the exact representation of certain components. For example, in many cases a one-to-one relationship implies that the element can be represented as an attribute (even perhaps a complex attribute) rather than entity. Furthermore, N-ary associations should be decomposed into binary (rarely ternary [29]) relationships, for a more realistic visualization of the database structure. Other similar steps must be considered as well.

An additional consideration to be added to the process is the earlier mentioned ability of Mnesia to store Erlang terms of arbitrary complexity in the attribute fields, for example a record that in itself would candidate for being an entity.

The goal of the described method is to obtain a visual representation of the database schema, the corresponding ER diagram of the most important components of the database, as well as getting acquainted with the overall structure and functioning of the database. The iterative approach allows to gradually add entities, based on their relevance to the identification of data constraints.

3.4 Identification of data constraints

In order to test the business logic, we need to find the data constraints. However, it is often the case that data constraints are not explicitly documented, and identifying them is not a trivial task. There have been several case studies focusing on the extraction of business rules from COBOL programs [21] and applications using object oriented databases [6]. Unfortunately these efforts resulted in very narrow automated solutions, suitable for the specific purpose of the respective studies. Therefore, in the current project we approach the identification of database constraints from two directions: an analysis of the database reverse engineered into a visual structure and individual in-depth interviews with several of the developers of the Kred application.

The analysis of the visual representation of the database structure focused on the key elements of the database structure, such as the schema tables and primary keys, entities within the ER diagram as well as the relationships between them. We used Chen's notation for entity-relationship modeling [13], particularly since in this notation *relationships* are first class objects and can thus have attributes of their own. The latter is important for modeling Mnesia databases, where the relationship between two tables can be expressed in a table containing several additional elements. We express such elements as attributes of the relationship in the entity-relationship diagram. The first step in identifying constraints is to note the primary and foreign keys of the entities in order to establish the relations between the entities. For example, if two entities have a relationship between them, and share a set of foreign keys (which are primary keys in other tables), we expect the values of the foreign keys to be equal in all cases. Multiplicity will not be a deciding factor in this process, since it cannot be precisely determined without additional semantic information. This approach will help identify a part of the referential integrity constraints within the application.

The identified constraints have been validated during a presentation to the system developers. Furthermore, interviews with the developers have identified additional semantic information determining domain specific data constraints. The goal has been to identify an initial set of constraints that were recognized by the developers, rather than identifying all of constraints, which would require a lot of effort for a non-trivial large scale system. The initial set has been used to perform testing and to evaluate whether our approach could find inconsistencies in the data.

The ER diagram, together with the database schema produced as a result of the above mentioned reverse engineering of the database should yield enough information to determine part of the constraints. We have identified the following two categories of data constraints: referential integrity constraints, and domain specific data constraints.

3.4.1 Referential integrity constraints

As mentioned above, Mnesia does not provide support for referential integrity constraints. Therefore, referential integrity should be embedded in the business logic implementation. Referential integrity checks are easiest to discover, by examining the ER and the schema representation of the database as previously described. An example of a referential constraint identified in the current project. It is expressed as an SQL query, and should return NULL in case the constraint is satisfied. This particular example describes the relation between the tables **ptrans** and **pbal**:

```
SELECT ‘ptrans‘.‘ano‘
FROM ptrans, pbal
WHERE
  ((‘ptrans‘.‘pbal_key‘ =‘pbal‘.‘key‘)
  AND NOT
   (‘ptrans‘.‘invno‘ =‘pbal‘.‘invno‘ ))
```

This example constraint ensures that the **pbal** events (i.e. events that influence the payments balance of the account) and the **ptrans** events (i.e. events that are related to a personal account but do not influence the payment balance of the account) refer to the same invoice number, in case the **ptrans** table contains the key of the pbal event. Since there is a simple direct relationship between the two tables, it is likely that the constraint has been identified and checked by the developers, hence the probability of revealing an inconsistency error is quite low.

3.4.2 Domain specific data constraints

Business rules are domain- and business-specific constraints which are expected to be expressed in the business logic. Identification of domain specific data constraints is difficult, especially in the situation when semantic information about the system is not available. This task requires a combina-

tion of the above mentioned analysis of the schema representation and ER model of the database, code analysis and finally interviews with the developers familiar with the system. Code analysis includes tracing the events generated by the execution of the interface functions, examination of event logs and static code review. Below follows an example constraint, which is similarly to the previous one expressed in SQL and should return NULL in case the constraint holds.

```
SELECT ‘pbal‘.‘key‘
FROM ‘pbal‘, ‘invoice‘
WHERE
  ((‘invoice‘.‘invno‘  = ‘pbal.invno‘)
  AND
   (‘invoice‘.flags‘ = ?FI_IS_PACC))
```

This slightly more complex domain specific business rule ensures that the invoices that have their 'pstatus' flag set to "?FI_IS_PACC", which means that they belong to a personal account and therefore should have at least one payment balance (pbal) entry. Certainly such a constraint should not be incorporated into the implementation of the DBMS and is therefore left to the business logic implementation.

3.5 Testing data constraints

Before actually testing the identified data constraints, they have first been validated by the developers familiar with the system. This is needed in order to avoid errors in formulating constraints as a result of the lack of familiarity with the system. The earlier mentioned methodology of Castro and Arts is used to test the data constraints. Below are the stages of the methodology, adapted to the specifics of the project. A more thorough description can be found in [4].

Since the system under test uses Mnesia as data storage, the identified data constraints will have to be converted to Query List Comprehensions (QLC), which is Mnesia's query language. This query is written as an invariant to validate that the business rules hold before and after test execution. For example, the business rule presented above is expressed using QLC as follows:

```
invariant_pbal() ->
   QH = (qlc:q([Pb#pbal.key ||
                Pb <- mnesia:table(pbal),
                Inv <- mnesia:table(invoice),
                Pb#pbal.invno == Inv#invoice.invno,
                Inv#invoice.flags == ?FI_IS_PACC])),
   {atomic, Response} =
       mnesia:transaction(fun() -> qlc:e(QH) end),
   Response /= [].
```

The state of the database will be checked against the invariant both at the start and end of the test case, thus ensuring that the business rule is respected. A very important aspect at this point is the correct design of the test cases that will be run. A few test cases selected by the developers will be insufficient, because of the developers assuming system constraints that may not hold. Instead, a large number of test cases, that are valid operations but are extremely unlikely to ever happen during system operation, has to be generated.

The next step will be to identify the available interface functions to modify the states of the system. Depending on the architecture and the implementation of the system under test, this stage can be very time consuming. In examining the choice of the interface functions it is important to note their position relative to the implementation of the business logic.

If the interface functions are determined, generators are written for sequences of interface calls to the system. The generators will produce only the minimal set of data which is needed for the interface calls, in order to produce valid state transitions. At the same time, the generated data sets should be as varying as possible, in order to explore any potential non standard behavior of the application.

We present a few of the generators we developed to show what they look like and how similar they are to Erlang functions. The following generator would generate lists of items that can be used as an argument to an interface function. The generator shown below will produce sequences of varying length containing fairly different item sets. The generators for *artno*, *vat* and *discount* will produce small natural numbers:

```
list(#item{artno = nat(),
           description = list(char()),
           vat = choose(nat),
           flags = 0,
           discount = nat(),
```

The generator for *price* will produce large numbers with two decimals. Finally, the generator for *quantity* will produce either very large values, or small values for the quantity parameter. This will produce values at the boundaries rather than obtaining a normal distribution of number, as the use of `choose/1` would yield.

```
price = ?LET({H,F},{nat(),  choose(0,99)},
             (H*100)+ F\ 100),
quantity = ?LET({N,T,I},{nat(),choose(0,1),largeint()},
                N + T*abs(I))})
```

It can be argued that the values of these sequences do not affect the business logic, and are artificial, hard coded sequence of goods would suffice. However, this depends on the implementation of the business rules and the price paid for random generation is extremely low.

For the selected interface functions, a local function is written in order to validate that the response from the interface function corresponds to the expected result. For example, the interface function via the `estore_server` module that is used to activate a reservation gets a local variant as follows:

```
activate_reservation(Reservation, Items, Pno) ->
  Result =
```

```
    estore_server:handler(
       'undefined',
       {call, activate_reservation,
       [Reservation, Items, Pno, (...)]}),
Person = person:read_d(Pno),
Blacklisted =  (Person#person.blacklisted == 1),
case Result of
     {false,{response,[{array, ["no_risk",Invno]}]}}
         when not Blacklisted -> Result;
     {false,{response,{fault, -4,"blocked"}}}
         when Blacklisted -> Result;
     _ -> exit(unexpected value)
end.
```

First the function is called and the result is stored, after that, the result is compared to the expected outcome.

After having added all the interface functions, QuickCheck will create test cases by running sequences of generated interface calls. The results will be validated through the expected values, and finally the invariant will be checked. A situation in which the invariant evaluates to 'false' would mean that the business rule has been invalidated, and the database is in an inconsistent state. The available test sequence will make it possible to observe the exact actions that have invalidated the data constraints. Furthermore, QuickCheck will automatically shrink the test sequence to a minimal failing case in order to show the exact cause of the error.

3.6 Analysis of the test results

The results collected by running the tests developed according to the above described methodology will be used to evaluate the way the business logic *actually* enforces the data constraints in contrast to the *expected* enforcement of data constraints. Furthermore, the data will be used to verify whether the approach is fully applicable to database applications which use non-relational unnormalized databases.

4. Results

4.1 Reverse engineering

One of the obtained results is the reverse engineered ER diagram and a raw representation of the schema of the database. The obtained ER diagram is a useful artifact for Kreditor, and together with the initial set of defined and formalized constraints will contribute to the current system documentation. At the same time, it is an essential document for our test approach, since we extract constraints from this ER diagram that we use to test against.

We used the data in the schema files, the table descriptions to obtain our first rough estimate of the ER diagram. A schema is a set of record definitions, each record has a name, the *table name*, and a number of fields, corresponding to the *table fields*. We initially assumed each record to correspond to an entity and the fields to attributes. After that we assume equal field names (attributes) to symbolize relations. That is, if a entity **pbal** has an attribute *invno* and the entity **ptrans** also has an attribute *invno*, then we assume that these entities are related and the attributes are replaced by a relation symbol. The kind of relation is unknown, it can be one-to-one, many-to-one, or something else, but that is impossible to infer from the schema file.

In the second iteration of the reverse engineering process, 18 of the entities were transformed into relationships. This was done in order to both reduce the complexity of the ER model, as well as bring the ER model close to the actual structure of the database. For example, the entity *personal_email* was converted to a relationship between

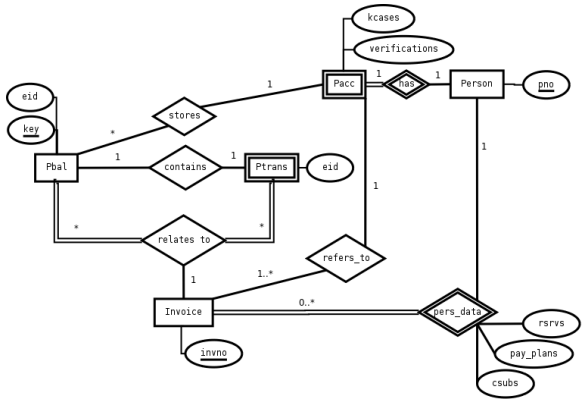

Figure 1. Fragment of the ER model of the database

estore and **person**. The other elements contained in the *personal_email* were noted as the attributes of this relationship.

The primary key of each table is assumed to be the first field of the record definition, since that's the standard in Mnesia. In this way we visualized the ER diagram using 43 of the 87 tables that the developers considered as most relevant.

Of course, we identified entities that had more than one attribute in common with each other. For example, the entity **pbal** and **ptrans** have 2 attributes in common: *invno* and *key*. Since *key* also occurred in a third entity, viz. **pacc**, we created two relations between the entities, as depicted in Figure 1.

There is, of course, a risk that certain attributes have the same name, but do not identify a relationship. Similarly, it may be the case that there is a relationship between fields that have different names. In our case for example the attributes *invno* and *ocr* expressed a relation, where *ocr* is a non-standard name for the invoice number. In addition, it is totally unclear what kind of relation the attributes symbolize. Therefore, we consulted the domain experts to look at the ER diagram and provide us with feedback.

This revealed a number of unclarities in the author's model of the database, for example the already mentioned relation hidden behind *invno* and *ocr*, but also more sophisticated issues. For example, the *Pno* which is used as an alias for *Personal Number* throughout the database implementation, can be used to denote both the Personal Number for physical persons, as well the Organization number for legal entities. Therefore, the relation between two entities is context dependent and a zero-to-many relationship.

After consulting the developers, the resulting ER diagram contained 23 entities and 36 relations and a total of 250 attributes. Obviously, attempting to discover all data constraints that can be found in the ER diagram would be a daunting task, therefore we only selected a subset of possible constraints for the five weeks we had left for our case study.

4.2 Constraints Identified

For the purpose of the project, 24 constraints have been identified and recorded for further testing. The constraints were initially expressed in SQL in order to be validated during individual interviews with developers. Unexpectedly, of the 24 identified constraints only 16 have been considered as valid, while the other 8 were considered as either irrelevant, or not true for the system.

The reason for such large number of invalid constraints can be explained either by the misinterpretations and miscellaneous errors in the process of reverse engineering the database, or by the often stated *lack of familiarity with the system*. However, the identification of invalid, or false constraints should not be considered as a waste of time. This is simply because formulating and discussing these *incorrect* constraints made it possible to both learn that not all relations are relevant at some point in the business process, despite their apparent semantic similarities.

In any case, even a reduced set of constraints is important, since no other constraints of the Kred application have been recorded earlier. The above mentioned constraint

```
SELECT ‘ptrans‘.‘ano‘
FROM ptrans, pbal
WHERE
 ((‘ptrans‘.‘pbal_key‘=‘pbal‘.‘key‘)
  AND NOT
  (‘ptrans‘.‘invno‘=‘pbal‘.‘invno‘ ))
```

was remarked as particularly relevant, since there have been situations in the past when this constraint was not respected.

4.2.1 Failing constraints

We used QuickCheck to generate random sequences of calls to the interface functions, or in other words, have users of the system "go wild on it. After each such sequence we validated the identified constraints. Surprising enough, we detected that two of them could be violated.

Contrary to the earlier expectations that the referential constraints are most likely to hold (in contrast with the domain specific business logic, the errors in which are more difficult to spot), the constraint provided earlier as an example, did not pass the test (the output details are ignored):

QuickCheck has found a counterexample when **ptrans.invno** is *not* equal to **pbal.invno**, when pbal.pno is equal with ptrans.pno. This data constraint has been detected through the analysis of the ER diagram and later confirmed by several developers as correct. However, in some *apparently* rare cases this referential integrity constraint does not hold. QuickCheck's shrinking technique made analysis of this rare case an easy task.

A second failing constraint that has been discovered was a "domain specific data constraint" (according to the above classification). It will not be described further, however its discovery demonstrates that the applied methodology allows us to discover both failing referential integrity data constraints, and domain specific business rules.

4.3 Constraints testing results

4.3.1 Validation of the test specifications

When constraints are violated by a sequence of interface calls, one needs to ensure oneself that indeed the constraint should hold and the sequence of calls introduces an error. When recognizing this, the error can be fixed and the same sequence can be executed again, now not resulting in a violation.

However, what do we know if a constraint is not violated? Probably we simply formulated a database query that is always satisfied and does not really describe the constraint we wanted to validate. For each constraint we also wanted to check that this constraint expressed what we intended. This is problem similar to ensuring that your test suite is correct. Several methods to do so exist.

- Mutation testing, which involves changing the source code of the system under test [36], [33].
- Fault injection, a method involving altering the source code to test code paths that might not be visited [3].
- The use and probation of various test design approaches: using classification trees [18], the Z method [20], or even a combination of the two [32].
- Deliberately alter a constraint so that it *must* fail, and test that it actually fails during test execution.
- Introduce a change in the implementation of the system – the change should produce a controlled fault that would invalidate a constraint (that otherwise holds) during test execution.

We believe that combining all (or several) of the described methods would yield the highest certainty that the test specifications are correct. However, this section will describe the application of the second method, and namely the deliberate introduction of a software fault that would invalidate a certain constraint during test execution.

In order to apply this approach, the following constraint has been chosen:

```
SELECT ‘invoice‘.‘pno‘
FROM ‘invoice‘, ‘estore_data‘
WHERE
 (‘invoice‘.‘eid‘ = ‘estore_data‘.‘eid‘
AND NOT
(‘invoice‘.‘pno‘ in ‘estore_data‘.‘customers‘))
```

This constraint ensures that whenever a new customer makes a purchase in the estore, they are added to the list of customers in the estore_data table of the corresponding estore.

This example has been chosen both for its simplicity (which becomes highly valuable in an unknown and complex system) and for the relatively low effort of adding a fault that would violate the constraint. To produce this fault, the code is altered to that the customers list of estore_data is emptied each time a new invoice is added. Though it might seem rather raw, the constraint is guaranteed to fail once there are some invoices added.

Once the tests are run, QuickCheck quickly spot an example sequence in which this property is violated. It might be worth noting that despite the apparent triviality of the described bug, other existing test suites did not discover it.

5. Discussion

The results of the project show that overall, the method described in the paper of Castro and Arts [4] and applied to the present case is easily extendable and applicable to application which use databases such as Mnesia. The quality and time efficiency of applying this methodology depends significantly on the level of documentation of the system, the application's complexity and the clarity of the application's implementation. The approach produces several positive outcomes, namely the updated ER model of the database, and a schema representation of some of the tables. The most important outcome however, is the set of specifications and formalized constraints that is available once this method has been applied. Such a test framework can be used (and continuously updated) later to ensure that the data constraints are always ensured when new functionality and components are being developed.

As mentioned above, there are two main steps in the methodology (performed iteratively), namely *identification of the constraints*, and *development of the test specifications*. There are several factors that influence the outcome of the test procedure using the above described method.

5.1 Available Documented Constraints

First of all, the availability of documented constraints would significantly facilitate the testing process. However, the fact that there is a set of documented constraints does not imply that one need not search for additional constraints. Considering that documentation can be outdated or incomplete, the constraints identification process should precede the constraints testing stage. Nevertheless, explicitly formulating the business constraints along the development of the system would greatly facilitate their later testing.

5.2 System Documentation

Availability of system documentation is also important when defining the data constraints and developing the test specifications. Semantic information, extracted out of the system documentation can add details to the ER model of the database in case it is developed through reverse engineering, or increase its understanding, in case it is readily available.

Furthermore, system implementation can help obtain the *domain knowledge* necessary for an effective detection of the data constraints. The experience of this project has shown, that a combination of absent documentation and insufficient domain knowledge can lead to a situation when 33% (8 out of 24) of the identified constraints will be unusable.

At the same time, absence of documentation is a fact of live and documentation easily gets outdated. The formalized constraints together with a QuickCheck framework are helpful in keeping the documentation alive, since changes in the program may make test cases fail.

5.3 Choice of Interfaces

A limitation encountered during the study was that the chosen XMLRPC interface did not provide access to the entire functionality of the Kred application. In the process of daily usage, the data within the application is modified through other existing system interfaces as well, for example the GUI. However, the effort required for the extra set up for the GUI testing was disproportionally high compared with the overall scope of the project. The inability to fully mimic all peculiarities of data handling during the tests has thus prevented us from a more thorough examination of the business logic. We can assume, that in a database application with multiple data access interfaces, a complete testing of the business logic also requires simultaneous testing of all available application interfaces.

In case that all, or most of the above conditions are fulfilled, the testing process can be focused on developing the test specifications. However, the current project has followed a different path and the following steps have been taken:

- Reverse engineer the database to obtain the database schema and ER model.
- Analyze the ER diagram to determine initial data constraints.
- Analyze the source code to identify other business logic constraints.
- Verify the obtained constraints with the developers who posses the domain knowledge about the system under test.
- Determine the interface functions that will be called in the testing process.
- Design test cases to test the data constraints.
- Implement the test case specifications using QuickCheck
- Run the tests and analyze the results.

6. Conclusions

In this case study we wanted to evaluate the methodology of Castro and Arts [4] for testing data consistency of data-intensive applications by examining a database application which uses an unnormalized non relational database. We have adopted a customized approach for extracting the data constraints by reverse engineering the database and expressing the constraints in a database-specific query language. Further, we have tested several interface functions with the QuickCheck testing tool and revealed a constraint violation.

There were several notable points in the process of constraint testing according to the adopted methodology. First, reverse engineering of the database structure is a crucial stage for the identification of data constraints. We have examined an unnormalized non relational database, and reverse engineered it according to a simplified version of the method described by Premerlany and Blaha [29] to obtain an ER diagram of the database. We have seen that important elements like multiplicity cannot be inferred without semantic information and can therefore affect the elicitation of database constraints.

The obtained ER model was used to extract and define data constraints that were present in the application. We have determined two types of constraints, namely referential constrains and business rules. Referential constraints can be identified by examining the ER model of the database and represent constraints based on foreign key relations between tables. We have seen that, despite our expectations and their relative discoverability referential integrity constraints can contain implementation faults, since we have found a violation of a referential constraint during the testing process.

On the other hand business rules cannot, in most of the cases, be identified through the examination of the database ER model, and therefore require the semantic knowledge of domain experts. We did not discover any violations of the business rules that have been tested. One of the reasons for this is the relatively small number of business rule constraints that were identified and tested. Another possible reason is the choice of system interface to be tested.

We have observed that the ratio of invalid or else incorrect constraints out of the total number of identified constraints was significantly higher in the first stages of the project, after the first iteration of database reverse engineering. Later on, the number of valid constraints has grown together with the understanding of the internals of the database application. Based on this, we can state that there might be a connection between the understanding of the application's implementation and the efficiency of the constraints elicitation process. On the other hand, improving and refining the process of constraints elicitation –both referential constraints and business rules – would be a topic for further research.

As mentioned above, in the current study we have examined a selection of data constraints and tested them with a limited number of interface functions. However, a complete elicitation of the data constraints present in the Kred application would require a revision and completion of the ER database model, further analysis of the relations between the entities in the model and interviews with domain experts.

We have limited ourselves and did not explore the effects of distribution and concurrency on the data constraints

within the application, despite both of them being important properties of Mnesia. Studying the effect of these two aspects can also be the topic of future research.

Taking into account the findings of the project, we can state that the adopted methodology could be applied to database applications which use non relational database management systems (particularly Mnesia), and unnormalized databases. We also contributed by applying the approach of Premerlany and Blaha to non relational databases and thus touching upon the topic of reverse engineering Mnesia databases.

Acknowledgments

The authors would like to thank everyone who has contributed to this paper with corrections, feedback and valuable input. Special thanks to the operational and software development teams at Kreditor for their help and support.

References

[1] M. Andersson, "Extracting an entity relationship schema from a relational database through reverse engineering," in *ER '94: Proceedings of the13th International Conference on the Entity-Relationship Approach*, (London, UK), pp. 403–419, Springer-Verlag, 1994.

[2] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Trans. Comput.*, vol. 42, no. 8, pp. 913–923, 1993.

[4] T. Arts and L. M. Castro, "Testing data consistency of data-intensive applications using quickcheck," Technical report ITU, to be published, 2009.

[5] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with Quviq QuickCheck," in *ERLANG '06: Proc. of the 2006 ACM SIGPLAN workshop on Erlang*, pp. 2–10, ACM, 2006.

[6] N. Bassiliades and I. P. Vlahavas, "Modelling constraints with exceptions in object-oriented databases," in *ER '94: Proc. of the13th Int. Conf. on the Entity-Relationship Approach*, (London, UK), pp. 189–204, Springer-Verlag, 1994.

[7] J. Blom and B. Jonsson, "Automated test generation for industrial erlang applications," in *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, (New York, NY, USA), pp. 8–14, ACM, 2003.

[8] M.-C. Boudreau, D. Gefen, and D. W. Straub, "Validation in information systems research: A state-of-the-art assessment," *MIS Quarterly*, vol. 25, no. 1, pp. 1–16, 2001.

[9] C. Calero, M. Piattini, and M. Genero, "Empirical validation of referential integrity metrics," *Information and Software Technology*, vol. 43, no. 15, pp. 949 – 957, 2001.

[10] M. Y. Chan and S. C. Cheung, "Testing database applications with sql semantics," in *In Proc. of the 2nd Int. Symp. on Cooperative Database Systems for Advanced Applications*, pp. 363–374, Springer, 1999.

[11] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in *QSIC '05: Proc. of the Fifth Int. Conf. on Quality Software*, (Washington, DC, USA), pp. 187–196, IEEE Computer Society, 2005.

[12] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An agenda for testing relational database applications: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 1, pp. 17–44, 2004.

[13] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.

[14] K. H. Davis and A. K. Arora, "Converting a relational database model into an entity-relationship model," in *Proc. of the Sixth Int. Conf. on Entity-Relationship Approach*, pp. 271–285, 1988.

[15] Y. Deng, P. Frankl, and D. Chays, "Testing database transactions with agenda," in *ICSE '05: Proceedings of the 27th int. conf. on Software engineering*, (New York, NY, USA), pp. 78–87, ACM, 2005.

[16] J. Dietrich and A. Paschke, "On the test-driven development and validation of business rules," in *Information Systems Technology and its Applications, 4th Int. Conf., 23-25 May, 2005, Palmerston North, New Zealand, volume 63 of LNI*, pp. 31–48, GI, 2005.

[17] A. Eisenberg and J. Melton, "Background sql:1999, formerly known as sql3," *Commun. ACM*, 2008.

[18] M. Grochtmann and D. benz Ag, "Test case design using classification trees," 1994.

[19] J.-L. Hainaut, "Database reverse engineering: Models, techniques and strategies," in *Proc. Of the 10 th Int. Conf. on Entity-Relationship Approach*.

[20] S. Helke, T. Neustupny, and T. Santen, "Automating test case generation from z specifications with isabelle," in *ZUM '97: Proc. of the 10th Int. Conf. of Z Users on The Z Formal Specification Notation*, (London, UK), pp. 52–71, Springer-Verlag, 1997.

[21] H. Huang, W.-T. Tsai, S. Bhattacharya, X. Chen, Y. Wang, and J. Sun, "Business rule extraction techniques for cobol programs," vol. 10, (New York, NY, USA), pp. 3–35, John Wiley & Sons, Inc., 1998.

[22] W. Kent, "A simple guide to five normal forms in relational database theory," *Commun. ACM*, vol. 26, no. 2, pp. 120–125, 1983.

[23] F. N. Kerlinger, *Foundations of Behavioral Research*. Harcourt Brace Jovanovich, 1986.

[24] M. Ketabchi, S. Mathur, T. Risch, and J. Chen, "Comparative analysis of rdbms and oodbms: a case study," in *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Comp. Soc. Int. Conf.*, (New York, NY, USA), pp. 528–537, IEEE, 1990.

[25] V. V. Kuliamin, "Model based testing of large-scale software: How can simple models help to test complex system," in *In Proc. of 1-st Int. Symp. on Leveraging Applications of Formal Methods*, pp. 311–316, 2004.

[26] H. Mattsson, H. Nilsson, and C. Wikstrom, “Mnesia - a distributed robust dbms for telecommunications applications,” in *PADL ’99: Proc. of the First Int. Workshop on Practical Aspects of Declarative Languages*, (London, UK), pp. 152–163, Springer-Verlag, 1998.

[27] J. Nielsen and V. L. Phillips, “Estimating the relative usability of two interfaces: heuristic, formal, and empirical methods compared,” in *CHI ’93: Proc. of the INTERACT ’93 and CHI ’93 conf. on Human factors in computing systems*, pp. 214–221, ACM, 1993.

[28] M. Phalippou and R. Castanet, “Relations d’implantation et hypotheses de test sur des automates a entrees et sorties = implementation relations and test hypotheses on input-output automata,” in *Travaux Universitaires - These nouveau doctorat*, (Universite de Bordeaux 1, Talence, FRANCE), 1994.

[29] W. J. Premerlani and M. R. Blaha, “An approach for reverse engineering of relational databases,” *Commun. ACM*, vol. 37, no. 5, pp. 42–ff., 1994.

[30] G. J. Tretmans and A. F. E. Belinfante, “Automatic testing with formal methods,” Technical Report TR-CTIT-99-17, Enschede, December 1999.

[31] J. Tretmans, P. Kars, and E. Brinksma, “Protocol conformance testing: A formal perspective on iso is-9646,” in *Proc. of the IFIP TC6/WG6.1 Fourth Int. Workshop on Protocol Test Systems IV*, (Amsterdam, The Netherlands), pp. 131–142, 1992.

[32] H. Singh, M. Conrad, and S. Sadeghipour, “Test case design based on z and the classification-tree method,” in *ICFEM ’97: Proc. of the 1st Int. Conf. on Formal Engineering Methods*, (Washington, DC, USA), p. 81, IEEE Computer Society, 1997.

[33] S. De Souza, D. R. S. Maldonado, J. C. Fabbri, S. Camargo, P. Ferraz, D. Souza, and W. Lopes, “Mutation testing applied to estelle specifications,” *Software Quality Control*, vol. 8, no. 4, pp. 285–301, 1999.

[34] I. O. for Standardization, “Information technology, open systems interconnection, conformance testing methodology and framework. international standard is-9646,” (Geneve, CH), p.290-294, ISO, 1991.

[35] G. project, “http://projects.gnome.org/dia/,” 2009.

[36] W. E. Wong and A. P. Mathur, “Reducing the cost of mutation testing: an empirical study,” *J. Syst. Softw.*, vol. 31, no. 3, pp. 185–196, 1995.