Constraint-based Test Database Generation for SQL Queries

Claudio de la Riva, María José Suárez-Cabal, Javier Tuya
Computer Science Department - University of Oviedo
Campus Universitario de Gijón – SPAIN
(34) 98518 2451
{claudio,cabal,tuya}@uniovi.es

ABSTRACT

Populating test databases with meaningful test data is a difficult task as it involves generating data for many joined tables that must be diverse enough to be able to reveal faults and small enough to make the testing process efficient. This paper proposes an approach for the automatic generation of a test database for a set of SQL queries using a test criterion specifically tailored for the SQL language (SQLFpc). Given as input a schema database and a set of test requirements derived from the application of the test criterion to the target queries, the approach returns a database instance which satisfies the test requirements. Both the schema and the test requirements are modeled in the Alloy language, after which the analyzer generates the test database. The approach is evaluated on a real case study and the results show its feasibility, generating a test database of reduced size with an elevated coverage and mutation score.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging - Testing tools

General Terms

Reliability, Experimentation, Languages, Verification.

Keywords

Software testing, database testing, test database generation, MCDC; SQL coverage, SQLFpc, Alloy toolset.

1. INTRODUCTION

Database applications play an important role in today's commercial systems. Most of the business logic of these applications is implemented using the SQL language and the testing process of the queries is a difficult task because it includes populating a test database which may involve many tables and then checking the results of executing the queries on the test database. In the area of the test database generation, there are a number of tools, for example [3], which enable the automatic generation of a database, but in general often ignore the semantics of the logic that should be tested and therefore do not exercise the queries.

Recent research introduces query-aware test generation procedures [1] [12] [5] [18], which use the information from both the query and the schema. In general, these approaches use very simple test criteria on the query to guide the test generation. However, due to the different semantics of SQL language compared to procedural languages, these criteria are not sufficient for testing SQL queries. On the other hand, there are a number of approaches in the literature, for example [10] [9] [21] [13] that focus the definition of test adequacy criteria for databases. These approaches are principally targeted to assess the adequacy of a test database, but not for test database generation.

In this paper, we address the issue of the query-aware test database generation for SQL queries. As test criteria, we use a test coverage criterion, named SQLFpc, specifically tailored to deal with the particularities of the SQL queries [17]. It is based on the Modified Condition Decision Coverage (MCDC) [6] and provides a set of test requirements or *coverage rules* that the test database must fulfill. Conceptually, given a query and a database schema, our goal is to automatically generate a test database that covers the test requirements specified by the SQLFpc criterion for the query. To do this, our approach first generates a set of test requirements from the target query, then models both the schema and the requirements as a set of constraints in the Alloy language [10] and finally uses the Alloy analyzer [22] to generate an instance that satisfies both the schema and the test requirements. The generated instance is a test database that covers many of the test requirements for the target query. Although the use of Alloy for database testing is not new [12], we provide a novel representation of the database schema by modeling all the tables in a single structure that enables the support for larger databases schemas.

On the other hand, it is usual to use the same test database for a set of queries, as it reduces the cost of the test preparation and execution. Unlike many query-aware generation procedures, which generate one test database for each individual query of an application, our approach supports the automatic generation of a single test database for multiple queries within the application.

The main contributions of this work are:

- **Query-aware Test Generation**. We present an approach for automatic populating test databases which employs a coverage criterion specifically tailored for SQL queries.

- **Alloy for Test Database Generation**. We propose a novel representation in Alloy that enables handling larger databases and multiples queries.
- **Evaluation**. We evaluate the approach on an industrial case study including a number of queries and a schema with a large number of tables and columns.

The remainder of the paper is organized as follows: Section 2 introduces the background and notation used in the paper. Section 3 describes in detail the test database generation using the Alloy toolset. Section 4 describes the results of the experiments over a real case study and Section 5 discusses the related work. The paper ends with conclusions and future work.

2. BACKGROUND AND NOTATION

2.1 The Relational Model

Relations. In the relational model of databases [7], given a set of attributes $A=\{A_1,A_2,..,A_n\}$ over a set of domains $D_1,D_2,\ldots,D_n$ a relation denoted as $R(A_1,A_2,\ldots,A_n)$ or R(A) is a subset of the Cartesian product of the domains. In the SQL language, a relation is a table, a tuple is a row of the table, attributes are the columns of the table and the domains define data types of the attributes. A characteristic of the relational model is the definition of the three-valued logic corresponding to the presence of missing information in the values of the attributes (null values). To handle the missing information, we define the Boolean predicate $nl(A_i)$ that is true at the tuples where the attribute A_i is null.

Database constraints. Database constraints specify restrictions on the database that require relations and attributes to satisfy certain properties. In SQL, the *nullability constraint* (NOT NULL) forces an attribute to not accept null values. An attribute is *nullable* if it has been declared in the database schema without the NOT NULL constraint. The *primary key* constraint specifies a set of attributes in a relation that uniquely identifies a tuple of the table. The *foreign key* constraint in a relation points to a primary key in another table.

Queries. Queries are defined in the form Z←rve, where rve denotes a *relation-valued expression* (an expression whose evaluation yields a relation) and Z is a relation containing the tuples obtained when applying the rve. The *select* operator Z←R[p(A)] (in SQL, `SELECT*FROM R WHERE p(A)`) uses a relation R(A) and generates a relation Z with some rows of R(A) according to the predicate p on the attributes of A. The *inner join* operator Z←R[p(A,B)]S (in SQL, `SELECT * FROM R INNER JOIN S ON p(A,B)`) uses two relations, R(A) and S(B), and generates a relation Z with tuples of R(A) concatenated with tuples of S(B) where the logical condition p(A,B) is evaluated true. The *left outer join* returns the result of the inner join, plus those tuples in R(A) that do not match the join operator. The *right outer join* operator is symmetric to the left join. The *full outer join* is defined as the union of the inner join, the left outer join and the right outer join. For identifying each join type in an rve expression, we use the notation $R[p(A,B)]^T S$, where T={I,L,R,F} denotes the join type (inner, left, right and full, respectively).

In this work, we support SQL SELECT queries consisting of WHERE conditions and a number of JOIN operators. Also, the nullability and the primary and the foreign key database constraints are supported.

Figure 1(a) shows an example of a database schema defining the `Orders` and `ItemLine` tables with their nullability, primary and foreign key constraints. Figure 1(b) shows an SQL query that lists the orders and their item lines for those item lines with quantity distinct from 5 or price equals 3. The rve for the query is $Orders[oid=ioid]^I ItemLine[quantity\neq5 \lor price=3]$. This example will be used in the rest of the work to illustrate the approach.

```
a)Database schema
CREATE TABLE Orders(
  oid INTEGER PRIMARY KEY NOT NULL,
  odate DATE);
CREATE TABLE ItemLine(
  iid INTEGER PRIMARY KEY NOT NULL,
  ioid INTEGER FOREIGN KEY REFERENCES Orders,
  quantity INTEGER,
  price INTEGER NOT NULL);
b)SQL query
SELECT * FROM
Orders O INNER JOIN ItemLine I ON O.oid=I.ioid
WHERE I.quantity <> 5 OR I.price = 3
```

Figure 1: Example of (a) database schema (b) query

2.2 Test Coverage Criterion for SQL Queries

An MCDC-based coverage criterion, named SQLFpc, for assessing the coverage of the test data in relation to a query that is executed is defined in [17]. It is specifically adapted for handling the particularities of the SQL queries. This criterion can also be used for designing meaningful test inputs. To illustrate this, consider the example query in Figure 1(b). Using this criterion, the tester should design a test database for covering the following test situations:

- Include rows such that the conditions in the WHERE clause `I.quantity<>5` and `I.price=3` are: **(1)** both false, **(2)** true and false and **(3)** false and true.
- Because the column `I.quantity` may be NULL, include rows such that **(4)** `I.quantity` is NULL and `I.price=3` is false.
- Because there is an INNER JOIN operator, include rows such that **(5)** there exists rows in the table `Orders` without joined rows in the table `ItemLine`, **(6)** there exists rows in the table `ItemLine` without joined rows in the table `Orders` and the WHERE condition is true. Note that this situation can be possible because `ioid` in the table `ItemLine` may contain the NULL value.

Each of these situations specifies a test requirement that can be expressed by an SQL query and constitutes a *coverage rule*. The process for generating a set of coverage rules from a given SQL query has been automated and implemented in the SQLFpc tool [23]. Applying this tool to the query of the example, we obtain the following coverage rules (expressed as relation-valued expressions and SQL queries) for the above test situations (2), (4) and (5):

(2) $Orders[oid=ioid]^I ItemLine[(quantity\neq5) \land\neg (price=3)]$

```
SELECT * FROM Orders O INNER JOIN ItemLine I
ON O.oid = I.ioid WHERE (I.quantity <> 5)
AND NOT(I.price = 3)
```

(4) Orders[oid=ioid]^{I}ItemLine[nl(quantity) $\wedge$ $\neg$(price=3)]

```
SELECT * FROM Orders O INNER JOIN ItemLine I
ON O.oid = I.ioid WHERE (I.quantity IS NULL)
AND NOT(I.price = 3)
```

(5) Orders[oid=ioid]^{L}ItemLine[nl(ioid) $\wedge$ $\neg$ nl(oid)]

```
SELECT * FROM Orders O LEFT JOIN ItemLine I
ON O.oid = I.ioid WHERE (I.oid IS NULL) AND
(O.oid IS NOT NULL)
```

Given a test database, a coverage rule holds if the execution of the corresponding SQL query against the test database produces at least one row as output. The coverage rules allow to measure the coverage of a test database against a set of queries or be used as a test input selection criterion. In the scope of this paper, they are used as a test selection criterion with the goal of populating the test database.

2.3 Alloy for Database Modeling and Testing

Alloy [10] is a first-order declarative language based on sets and relations. An Alloy model consists in a set of signatures (`sig`) which enables the definition of elements and a set of relations (fields) between them, and a set of constraints that restricts the space of the model. Constraints in Alloy can be of two types. A fact (`fact`) is a constraint that always holds. Predicates (`pred`) are constraints formulas whose satisfiability needs to be checked. The Alloy analyzer [22] translates the specification into propositional formulas and generates a set of finite scope instances that satisfy the constraints.

A previous approach in test database generation with Alloy [12] is based on modeling each table of the schema as a signature consisting of a relation over the domains of each attribute, that is, each table is modeled with a relation defined as the Cartesian product of its column domains (for a table with n columns, a n-arity relation). Database constraints are specified as fact constraints over the signature of each table. Although this representation is consistent with the relational model, it has the main drawback of its scalability. Because each table of the database is represented as the Cartesian product of its column domains and the analyzer enumerates all the solutions (test databases) that satisfy the specification, it could be infeasible to process large tables as the state space to analyze could be much larger. In our preliminary experiments, the Alloy analyzer ran out of memory in databases with tables with more than five columns (5-arity relations).

So, the approach here will be different. Instead of modeling the input schema (with a set of signatures with n-arity relations, one for each table), we will model the output of the coverage rules by means of a single signature consisting of a set of binary relations representing the attributes of the input schema. Details will be given in the next sections.

3. TEST INPUT GENERATION

Overview and Example. The approach takes as input a database schema and a set of SQLFpc coverage rules obtained from the target SQL queries and generates a database instance for testing the queries. Because the goal is generate a test database that fulfills the coverage rules, we model the database considering the output of the coverage rules, that is, a single relation consisting of all the attributes in the input schema. We name this relation as *database relation*. Given that a coverage rule holds if it execution produces at least one tuple in the output, coverage rules will be specified as constraints over the database relation with the aim to instruct the solver to find a solution over such relation. Finally, the tables of the input schema will be populated with this solution.

Consider the schema and the query of Figure 1 and the SQLFpc coverage rules in Section 2.2. The database relation is a relation consisting of the attributes of the `Orders` and `ItemLine` tables, plus an attribute `index` for identifying each tuple. This relation can be view as:

	Orders		**ItemLine**			
index	**oid**	**odate**	**iid**	**ioid**	**quantity**	**price**
…	…		…	…	…	…

In order to keep the data consistent with the relational model, we also must define a set of integrity constraints over this structure. For example, regarding the `oid` column that is the primary key in the `Orders` table, we must state that if its value is repeated in more than one tuple in the database relation, the corresponding values of the columns of the `Orders` table must have the same value.

Each SQLFpc coverage rule is defined as a constraint that the target database relation must satisfy. For example, the coverage rule corresponding to the test situation (2) (Orders[oid=ioid]^{I}ItemLine[(quantity$\neq$5)$\wedge\neg$(price=3)]) in the Section 2.2 is specified as a constraint with the aim of instructing the solver to generate at least one tuple in the database relation where (oid=ioid) and (quantity$\neq$5) and $\neg$(price=3). Figure 2 shows a possible solution that the solver could find considering the coverage rules (2), (4) and (5) of the example (Section 2.2). Finally, we populate each table of the schema using the solution found by the analyzer (see Figure 3). As a result, the generated database contains test data covering the test situations (coverage rules) for the query.

		Orders		**ItemLine**			
Cov. Rule	**index**	**oid**	**odate**	**iid**	**ioid**	**quantity**	**price**
2	1	1	date0	1	1	6	2
4	2	2	date1	2	2	NULL	2
5	3	3	date0	NULL	NULL	NULL	NULL

Figure 2: Example of a solution

Orders	
oid	**odate**
1	date0
2	date1
3	date0

ItemLine			
iid	**ioid**	**quantity**	**price**
1	1	6	2
2	2	NULL	2

Figure 3: Test database example

Problem Statement. Given a database schema $\mathcal{X}$, an SQL query Q and a set of SQLFpc coverage rules Δ(Q) for the query Q, the goal is to obtain an instance of the database for the schema $\mathcal{X}$ such that fulfills the coverage rules in Δ(Q). To make it possible, the approach proceeds with the following steps:

1) Model both the schema and the SQLFpc coverage rules as a set of constraints. The schema $\mathcal{X}$ is represented considering the output of the coverage rules, denoted as the *database relation* $\mathcal{R}$, and each coverage rule $\Delta_i \in \Delta$(Q) is specified as a constraint formula ∂_i that $\mathcal{R}$ must satisfy.
2) Solve the model in order to find an instance of $\mathcal{R}$ such that the coverage rules are fulfilled.

3) Populate each table of the schema $\mathcal{X}$ with the instance found to load $\mathcal{R}$.

In the following sections, we show how to model the database relation (and its integrity constraints) and the coverage rules in Alloy specifications, and how to find a test database that satisfies the coverage rules. We illustrate it through templates and examples that should be adequate for understanding how the approach works.

3.1 Database Relation Model

Given a database schema $\mathcal{X}$, the database relation $\mathcal{R}$ is a set of binary relations named *attribute relations* (one for each attribute in $\mathcal{X}$) in the form AttName(Num,AttDomain), where AttName is the name of the attribute, Num represents a domain used for identifying each element of $\mathcal{R}$ and AttDomain is the domain of the attribute. (ix,value)∈AttName denotes the value of the attribute AttName in the tuple identified by ix in $\mathcal{R}$. For example, the first tuple in Figure 2 is <(1,1),(1,date0),(1,1),(1,1),(1,6),(1,2)>. Because there is a mapping between an attribute in $\mathcal{X}$ and its corresponding attribute relation in $\mathcal{R}$, we will also use the term attribute to denote the attribute relation in $\mathcal{R}$.

The next template models the database relation $\mathcal{R}$ in Alloy, considering a database schema $\mathcal{X}$ formed by two relations, R(A) and S(B):

```
sig D1{} … sig Dk{} // Data types definition
sig Num {}
one sig NULL {} //denotes the null value
one sig DBrel{  //Database relation
  index: set {Num},
  // Attribute relations for R(A) and S(B)
  A1: Num -> {Di+NULL},…, An: Num -> {Di+NULL},
  B1: Num -> {Di+NULL},…, Bm: Num -> {Di+NULL}
}
```

Since Alloy only includes the basic type integer with a reduced scope, attribute domains are represented symbolically by defining an empty signature for each attribute domain. An additional data type with one element (NULL) is defined for representing the missing information. The database relation is represented by a single `sig` declaration (`DBrel`) that considers all the attribute relations, plus a field (`index`) for identifying each tuple in $\mathcal{R}$. Attribute relations are modeled as the Cartesian product between the index domain and the attribute domain. Because the null value could appear in each attribute value, the image of each attribute relation is extended with the NULL data type ({D_i+NULL}). To access the value of a given attribute A_i corresponding to the index x in `DBrel`, either the expression A_i[x] or x.A_i could be used.

3.2 Database Relation Constraints

A set of constraints must be defined in order to assure the consistency of the data in the $\mathcal{R}$ with regard to the relational schema $\mathcal{X}$. Because integrity constraints define restrictions in the database relation that is assumed to always hold and are applied over all tuples in the database relation, they are specified as a set of universal quantified fact constraints over the `DBRel` signature.

Index and Attribute Values. The first constraint that we must define is related to the values that the `index` could take in $\mathcal{R}$. Thus, for each tuple in $\mathcal{R}$, the value of the `index` must be unique (`all x:index | one x`). In the same way and for each attribute A_i in $\mathcal{R}$, the values in the domain of the attribute relation must also be unique (`all x:Ai.(Di+NULL)|one x.Ai`). In addition, for each attribute, we must impose that the values both in the index attribute and in the domain of each attribute relation are consistent between them. That is, each value in the `index` points to the domain of an attribute relation and vice versa:

```
all x: index | x in Ai.(Di+NULL)
all x: Ai.(Di+NULL) | x in index
```

Nullability Constraints. An initial idea could be to define a constraint for each attribute that is labeled with the NOT NULL in $\mathcal{X}$. However, in the database relation $\mathcal{R}$, this constraint could be not valid. Consider the example in Figure 2, row 3, where the `iid` attribute has the null value. The constraint does not hold, but the situation is feasible because it denotes a row in the table `Orders` without joined rows in the table `ItemLine` (see Figure 3). Thus, instead of formulating the nullability constraints for each individual attribute we define the following constraint rules:

- If an attribute is primary key in a relation of $\mathcal{X}$ and takes null values in a tuple in $\mathcal{R}$, all the attributes of the same relation must have the null value in the same tuple in $\mathcal{R}$.
- If an attribute is primary key in a relation of $\mathcal{X}$ and takes not null value in a tuple in $\mathcal{R}$, all the attributes of the same relation must have the not null value in the same tuple, excluding those attributes that are nullables, which can take any value (NULL or NOT NULL).

The following Alloy constraints define the above rules for the attribute relations corresponding to the attributes A_1,..,A_n in $\mathcal{X}$ assuming that A_1 is a primary key and the attribute A_n is nullable. For checking whether the data is null or not two predicates are defined (`isNULL` and `isNotNULL` respectively):

```
all x:index | isNULL[A1[x]]=>
isNULL[A2[x]] and … and isNULL[An[x]]
all x:index | isNotNULL[A1[x]]=>
isNotNULL[A2[x]]and … and isNotNULL [An-1 [x]]
```

Primary Key Constraints. The primary key constraints could be defined for each primary key of the relations in $\mathcal{X}$ by means of a fact constraint expressing that its value is unique. However, in $\mathcal{R}$ two or more tuples could have the same value for a primary key defined in the relations of $\mathcal{X}$. This situation occurs when a row in a table is joined with two or more rows in another table.

In $\mathcal{R}$ this type of constraints must state that if an attribute is a primary key in a relation of $\mathcal{X}$ and its value is repeated in more than one tuple, the corresponding values of the attributes of the relation must have the same value. Considering the attributes A_1 and A_2, where A_1 is a primary key and A_2 is nullable, the constraint in $\mathcal{R}$ is defined as follows:

```
all x: index,y:index |(NOT[eq[x,y]])=True and
                eq[A1[x],A1[y]]= True ) =>
                eq[A2[x],A2[y]] = True or
               (isNULL[A2[x]]and isNULL[A2[y]] )
```

The function `eq` implements the equal operator considering the characteristic of three valued logic (in a similar form, the function `neq` can be defined):

```
fun eq [x:univ,y:univ]:(Bool+ NULL){
  (isNotNULL[x] and isNotNULL[y]) =>
  (x=y => True else False) else NULL
}
```

Foreign Key Constraints. This applies to those attributes in a relation of $\mathcal{X}$ that reference other attributes in another relation. It states that the values corresponding to the foreign key attribute in a relation must be either a subset of those corresponding to the primary key attribute in the relation that the foreign key references or its value must be null if it is nullable. Given the schema $\mathcal{X}$, let B_1 be as a foreign key in the relation S(B) and A_1 the primary key in the relation R(A) that it references. The foreign key constraint in $\mathcal{R}$ is formulated in Alloy as follows:

```
all x:index | B1[x] in A1[num] or isNULL[B1[x]]
```

3.3 Coverage Rules Representation

A coverage rule for an SQL query denotes a test situation that the target test database must fulfill. Because a coverage rule holds if its execution over the test database returns at least one row, the goal is that there exist some tuples in the database relation that satisfy the coverage rules. Therefore, in addition to schema constraints, we must provide a constraint in order to satisfy each coverage rule. For each coverage rule $\Delta_i \in \Delta(Q)$ a predicate constraint ∂_i is defined. Since the goal is to instruct the analyzer to find at least one tuple satisfying the coverage rule, the corresponding constraints must be existentially quantified. For readability, here we restrict the presentation to coverage rules in the form $R[p(A,B)]^T S[q(A,B)]$, that is, an expression with a JOIN operator of type T and a condition p(A,B) and a WHERE clause with a condition q(A,B). For rules without JOINs or with nested JOINs, the representation is similar.

Coverage Rules with INNER JOIN. In general, given a database relation $\mathcal{R}$ and a coverage rule $R[p(A,B)]^T S[q(A,B)]$, the corresponding formula ∂ would be declared as follows:

```
some x:DBRel.index | p(DBRel.A, DBRel.B) and
                     q(DBRel.A, DBRel.B)
```

This predicate expresses that there must exist some tuples (`some x:DBrel.index`) such as both the JOIN predicate (`p(DBrel.A,DBRel.B)`) and the WHERE predicate (`q(DBrel.A,DBrel.B)`) hold. To model the constants in the WHERE predicate, we add new singleton signatures related with each constant defined in the coverage rule. For example, consider the SQLFpc coverage rule (2) of the example $Orders[oid=ioid]^I ItemLine[(quantity \neq 5) \wedge \neg(price=3)$:

```
some x: DBrel.index |
( eq[DBrel.oid[x], DBrel.ioid[x]] = True and
  neq[DBrel.quantity[x],const_5] = True and
  NOT (eq[DBrel.price[x],const_3] = True
)
```

This formula is sound when the join type is INNER, because it imposes that there exist tuples in the relation R(A) joined with tuples in S(B). However, for the OUTER JOINs, the formula must be modified.

Coverage Rules with OUTER JOINs.. In the case of LEFT JOIN, this coverage rule denotes a test situation where there exists at least one tuple in R(A) without joined tuples in S(B). The predicate that codes this constraint is:

```
some x: DBrel.index |
( p(DBrel.A, DBrel.B) or
  DBrel.A[x] !in DBrel.B[num] or
  isNULL[DBrel.A[x]] ) and q(A,B)
```

In this case, the JOIN predicate is "relaxed" allowing that the value of the primary key attribute in the relation R(A) is not referenced in the corresponding foreign key attribute in the relation S(B) (`DBrel.A[x]!in DBrel.B[num]`) or it has the null value (`isNULL[DBrel.A[x]]`).

For coverage rules with RIGHT JOIN, the procedure is symmetric.

3.4 Populating the Test Database and Support for Multiples Queries

In order to generate the target test database for a query Q that satisfies all the coverage rules $\Delta(Q)$, we must define a formula $\partial(Q)$ as the conjunction of the formulas for each coverage rule $\Delta_i \in \Delta(Q)$, that is, $\partial(Q)=\wedge\partial_i$. In Alloy, this formula is represented as a predicate encapsulating all the coverage rule constraints:

```
pred testDatabase {
   ∂1 and ∂2 and … and ∂n
}
```

Under certain circumstances it would not be feasible to find a single instance satisfying all constraints due to incompatibilities between two or more coverage rules. In order to detect and avoid this situation, the coverage rules are added to the `testDatabase` predicate and then executed in an incremental way; first, only the constraints ∂_1 and ∂_2 are considered and executed, then the constraint ∂_3 is added and so on. If the solver can't find a solution in each step, the last added coverage rule is discarded. As final result, only consistent coverage rules are in the `testDatabase` predicate. Future work will be addressed in order to achieve early detection of these incompatibilities prior to the solver execution.

Then, the analyzer is instructed (`run testDatabase`) to find an instance with symbolic values that fulfills both the database constraints and the `testDatabase` predicate. As an example, consider the output generated by the analyzer represented in Figure 2 (only the sets and relations corresponding to the `index` and the columns of the `ItemLine` table are presented):

```
index:{num$1, num$2, num$3}
iid:{<num$1,intType$1>,<num$2,intType$2>,
     <num$3,null$0> }
ioid:{<num$1,intType$1>,<num$2,intType$2>,
     <num$3,null$0>}
quantity:{<num$1,intType$6>,<num$2,null$0>,
     <num$3,null$0>}
price:{<num$1,intType$2>,<num$2,intType$2>,
     <num$3,null$0>}
```

In order to generate the specific test database, we assign the values of the attributes in each relation defined in $\mathcal{X}$ with the values of each attribute in the database relation $\mathcal{R}$ (`DBrel`) as indicated in the steps below:

1. For each relation in $\mathcal{X}$ and for each tuple in $\mathcal{R}$, extract the values of its constituent attributes in $\mathcal{R}$ and create a row for the corresponding table. If the row was previously created it will be redundant and therefore discarded.

2. Each symbolic value in the rows of a table must be mapped to the specific one, according to the data type defined in the schema definition.
3. Populate each table of $\mathcal{X}$ with the corresponding created rows by means of executing a set of SQL `INSERT` commands.

In the above example, the following rows for the `ItemLine` table are created:

```
<intType$1, intType$1, intType$6, intType$2>
<intType$2, intType$2, null$0, intType$2>
```

Each symbolic value is replaced by the specific values indicated in Figure 3, table `ItemLine`, and the rows are inserted in the table.

Multiples Queries. Up to this point, we have presented the approach for generating a test database considering only a target SQL query. Since the approach models the entire schema (all the tables) and not only the tables that participate in a specific query, conceptually it is also focused to generate a single database, but considering a number of SQL queries. Given a database relation $\mathcal{R}$, and a set of queries $Q_1,Q_2,\ldots,Q_n$ over $\mathcal{R}$, the formula $\bigwedge\partial(Q_i)$ specifies a constraint predicate enforcing to find an instance of $\mathcal{R}$ that satisfies the coverage rules for all the queries. As results, the analyzer returns an instance of $\mathcal{R}$ satisfying the coverage rules of all the queries, and therefore it is a test database for all the considered queries. In Alloy, the procedure is similar to that indicated above, but encapsulating the coverage rule formulas corresponding to the set of queries to be tested into the single predicate `testDatabase`.

```
pred testDatabase {
   ∂(Q1) and ∂(Q2) and … and ∂(Qn)
}
```

4. CASE STUDY

In order to illustrate the approach, a case study is presented where a test database is generated for a set of queries obtained from a real application. This application is a real-life helpdesk system that manages user requests. The main information stored is the helpdesk ticket, which is created for each user request. Whenever an action is performed on a ticket, a history record is created. The application implements a complete security subsystem that, before starting each transaction, executes a set of the SQL queries embedded in the procedural code.

The case study database to be populated is composed of 37 tables with 230 columns in all. Experiments have been run on an Intel® Core ™2 Duo PC, 2GHz with 3GB of memory and using the Alloy toolset v.4.0 and the SQLFpc tool v.1.0.63.0.

4.1 Test Database Generation

In this case study, 20 SQL queries have been selected with different complexity (number of joins and conditions in the WHERE clause). Following the process described, the set of coverage rules for all SQL queries is obtained applying the SQLFpc tool. Then, an Alloy specification is generated for the database schema and the set of coverage rules of all the queries under study. The resulting final Alloy specification contains a single signature (DBrel) for the database relation with 231 fields corresponding to the 230 columns of the input schema plus the index field, 1,347 database constraints (facts) and a unique predicate consisting of the 75 constraints corresponding to the coverage rules for all the queries.

The analyzer, with this specification, was instructed to generate an instance of the database. The analyzer returns a test database that satisfies 65 coverage rules out of a total of 75. There are another 10 coverage rules which have not been covered because they are inconsistent with the rest of the coverage rules and therefore do not allow the analyzer to find a solution. After examining the cause of this inconsistence, we found that in most cases, they were originated by incompatibilities between the WHERE and/or JOIN conditions in two or more coverage rules related to the same SQL query. For instance, consider the following two coverage rules of the case study:

```
SELECT ID from Status WHERE (Final=0) AND (ID=2)
SELECT ID from Status WHERE (Final<>0) AND (ID=2)
```

These rules state for test data in the `Status` table with the column `ID` equal to `2` and the `Final` column equal to `0` and distinct to `0` respectively. Because the `ID` is the primary key column, it is not possible to cover the two coverage rules in the same instance of the `Status` table and as a consequence test data is generated for a single coverage rule (the other rule is inconsistent).

The number of generated rows in the final test database was 139 spread over 32 tables. The rest of the tables (only 5) neither participate directly in the queries nor indirectly by means of foreign keys. For this reason, the analyzer does not have any constraint to assign values. The total time for generating the target test database, including the time needed for generating the SQLFpc coverage rules, solving all the constraints (database constraints plus those derived from the coverage rules) and executing the SQL commands to populate each table of the database was 285 seconds.

4.2 Analysis of the Results

In order to compare the fault detection ability and the coverage of generated test database, we use a copy of the production database. The production database has 22,387 tickets, 103,553 history records and 279 users. In total, the database contains 139,259 rows. The analysis of the results is considered from two aspects. Firstly, the percentages of SQLFpc coverage for both databases (the generated test and the production) were measured. To compute the coverage, the SQLFpc tool is used. Secondly, a mutation analysis is used to compare the effectiveness in detecting faults of both databases (test and production). In this case study, the mutation operators used are those defined by Tuya et al. [16] in order to check kinds of faults specifically related to the SQL language. For each of the queries in the case study, a set of mutated queries has been generated in an automated way using the SQLMutation tool [15] [24] and executed against the generated test and production database. The results of the two analyses are summarized in Table 1.

The ID column identifies each of the queries of the study. The second group of columns contains the results of the evaluation of the coverage: number of SQLFpc coverage rules (#CovR) for each query (75 coverage rules in total) and the percentage of covered rules (%SQLFpc) according to the SQLFpc criterion using the production database (Prod.) and the generated test database

(Test). For all the queries, the coverage using the generated test database is always equal or higher than using the production database. Therefore, with less data (139 rows in the test database, 139,259 rows in the production database), the coverage is greater (an average of 86.67% and 57.33% with the test and the production databases respectively). The last group of columns contains the results of the mutation analysis (#Mut. and %MutScore). The number of mutants in total is 3,479. As can be observed, in most of the cases, for all the queries of the study, the mutation score is greater in the test database than in the production database. On average, using the production database 66.54% of the mutants are dead, whereas with the generated test database the mutation score reaches an average of 84.13%. Therefore, the generated test database is more effective detecting faults even though the number of rows is far fewer.

Table 1: Results of SQLFpc coverage and mutation score using the production database and the generated test database

ID	#CovR	%SQLFpc		# Mut.	%Mut.Score	
		Prod.	Test		Prod.	Test
Q1	11	27.27	100.00	317	8.83	94.64
Q2	5	40.00	100.00	138	76.81	84.78
Q3	6	50.00	66.67	132	90.15	81.06
Q4	3	66.67	100.00	99	81.82	89.90
Q5	3	100.00	100.00	65	84.62	66.15
Q6	2	100.00	100.00	38	86.84	92.11
Q7	2	100.00	100.00	313	95.21	90.10
Q8	3	66.67	100.00	60	88.33	88.33
Q9	5	40.00	40.00	59	28.81	32.20
Q10	3	66.67	66.67	71	19.72	85.92
Q11	6	50.00	100.00	82	58.54	91.46
Q12	2	100.00	100.00	30	90.00	73.33
Q13	2	100.00	100.00	30	86.67	90.00
Q14	2	50.00	100.00	41	9.76	80.49
Q15	2	50.00	100.00	41	9.76	80.49
Q16	3	0.00	100.00	44	2.27	75.00
Q17	3	100.00	100.00	776	72.04	88.02
Q18	3	100.00	100.00	776	80.03	90.46
Q19	3	66.67	66.67	210	92.86	90.00
Q20	6	50.00	50.00	157	16.56	15.29
Total	**75**	**57.33**	**86.67**	**3,479**	**66.54**	**84.13**

In conclusion, the results of these experiments show the feasibility of the approach in order to populate a single test database for a set of SQL queries attaining good scores in the coverage and the fault detection ability. That implies that the generated test database contains a good diverse set of rows (in the sense that they exercise the target queries) that are good enough to be used for testing purposes (regarding fault detection capability).

5. RELATED WORK

Test database generation is a challenging problem which has concentrated some research efforts.

Some of them employ automated reasoning to populate the database. Khalek et al. [12] define a tool for data generation incorporating Alloy specifications both for the schema and the query. Each table is modeled as a separate n-arity relation over the attribute domains and the query is specified as a set of constraints that models the condition in the WHERE clause. As test criterion, they use a predicate coverage criterion over the predicate of the WHERE clause. However, this approach cannot handle tables with a larger number of attributes due to the arity of the table relations. In contrast with this work, our approach models the schema with a unique structure with a set binary relation, thus facilitating the support for larger schemas and databases, and we use a specific test criterion for SQL queries that takes into account the particularities of the SQL language, such as JOIN operators and nullable values. In Veanes et al. [18], the satisfiability modulo theories solver Z3 has been used to generate input data for SQL queries satisfying a given test condition. Whereas in this work the test conditions are given in an ad-hoc fashion (the query result is empty, nonempty, contains a value, etc.), our approach employs automated and query-based test conditions (SQLFpc coverage rules) to guide the database generation. Binning et al [1] propose a technique named "reverse query processing" for generating test databases that takes the query and the desired output as input and generates a database instance (using a model checker) that could produce that output for the query. This approach supports one SQL query and therefore generates one test database for each query. A further extension to this work [2] supports a set of queries and allows to specify to the user the output constraints in the form of SQL queries. However, the creation of these constraints could be difficult if the source specification is not complete. There are other works which use general purpose constraint solvers to populate the test database [14] [19] [8]. As in preceding works, the coverage criterion for generating the test database is not specifically tailored for SQL queries but rather for predicates or user constraints and therefore, the generated test database does not provide enough confidence to exercise the target query from a testing point of view.

In the specification-based area, Chays et al. [4] describe a tool that populates databases with test data that satisfies the schema constraints. However, in some situations the approach could return empty outputs. An improved approach is presented in [5] by means of the execution of "test generation queries". It requires tester intervention to provide the data groups whereas our work relies on the SQLFpc coverage rules and thus the test input generation process is fully automatic. Wilmor and Embury [19] develop a technique to specify intensional test cases for database applications. The database test cases are formed by preconditions that specify the initial state of the database and postconditions that must hold after execution of the target program. While the postconditions are outside the scope of this work, the preconditions have a similar purpose to the SQLFpc coverage rules we use, but in our approach they are automatically generated.

6. CONCLUSIONS AND FUTURE WORK

This work presents an approach for the automatic generation of test databases for a representative subset of SQL queries using a specific test criterion adapted for such queries and the Alloy toolset for modeling and data generation. Database schema is represented by means of a view of the entire database. Although this representation implies the definition of additional data consistency constraints with regard to those defined in the relational model, it does not introduce a significant overhead on the analyzer (in terms of time and space). On the contrary, we found that it facilitates the handling of larger databases.

The overall approach is fully automated so the tester is not required during the test data preparation phase. The experiments executed over a real case study show the feasibility of the approach in generating a single test database of reduced size with a high coverage and fault detection ability (measured in terms of the mutation score) for a number of non-trivial SQL queries over a large schema.

A practical limitation of the approach is derived from the limitations of the Alloy solver to handle arithmetical and string-based expressions that hinder its application to SQL queries with aggregate functions and string operations. We are investigating the use of the integer and string solvers with Alloy in order to support this type of clauses. In this line, we will extend the approach to handle other SQL clauses, such as subqueries, nested join and grouping and other SQL queries such as updates. In its current form the approach does not detect coverage rules incompatibilities until the solver is executed. The early detection of such inconsistencies between the coverage rules could help to increment the coverage of the generated test databases.

7. ACKNOWLEDGMENTS

This work has been funded by the Department of Science and Innovation (Spain) and ERDF funds (TIN2007-67843-C06-01) and the Government of Castilla-La Mancha (SV-09-UCLM-1) (PAC08-121-1374).

8. REFERENCES

[1] Binnig, C., Kossmann, D., Lo, E. 2007. Reverse query processing. In Proceedings of the 23rd International Conference on Data Engineering. IEEE Computer Society,Washington, DC, 506-515.

[2] Binnig, C., Kossmann, D., Lo, E. 2008. MultiRQP - Generating test databases for the functional testing of OLTP applications. In Proceedings of the 1st International Workshop on Testing Database Systems. DBTest ’08. ACM New York, NY, 1-6.

[3] Bruno, N., Chaudhuri, S. 2005. Flexible database generators. In Proceedings of the 31st International Conference on Very Large Data Bases.VLDB Endowment, 1097-1107.

[4] Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I., and Weyuker, E. J. 2004. An AGENDA for testing relational database applications. Softw. Test. Verif. Reliab. 14, 1 (Mar. 2004), 17-44.

[5] Chays, D., Shahid, J., Frankl, P.G. 2008. Query-based test generation for database applications. In Proceedings of the 1st International Workshop on Testing Database Systems. DBTest ’08. ACM New York, NY, 1-6.

[6] Chilenski, J.J. 2001. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, U.S. Department of Transportation, Federal Aviation Administration.

[7] Codd, E.F. 1990. The Relational model for database management - Version 2. Addison-Wesley.

[8] Emmi, M., Majumdar, R., Sen, K. 2007. Dynamic test input generation of database applications”. In Proceedings of the International Symposium on Software Testing and Analysis, ACM New York, NY, 151-162.

[9] Halfond, W.G.J., Orso. A. 2006. Command-form coverage for testing database applications. In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, Washington, DC, 69-80.

[10] Jackson, D. 2002. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11, 2 (Apr. 2002), 256-290.

[11] Kapfhammer, G. M. and Soffa, M. L. 2003. A family of test adequacy criteria for database-driven applications. In Proceedings of the 9th European Software Engineering Conference. ESEC/FSE-11. ACM, New York, NY, 98-107.

[12] Khalek, S.A., Elkarablie, B., Laleye, Y.O., Khurshid, A. 2008. Query-aware test generation using a relational constraint solver. In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, Washington, DC, 238-247.

[13] Suárez-Cabal, M.J., Tuya, J. 2009. Structural coverage criteria for testing SQL queries. J. Univers. Comput. Sci., 15, 3, 584-619.

[14] Tsai, W.T., Volovik, D., Keefe. T.F. 1990. Automated test case generation for programs specified by relational algebra queries. IEEE Trans. Software Eng. 16, 3, 316-324.

[15] Tuya, J., Suárez-Cabal, M.J., de la Riva, C. 2006. SQLMutation: A tool to generate mutants of SQL database queries. In Proceedings of the 2nd Workshop on Mutation Analysis. IEEE Computer Society, Washington, DC, 1-5.

[16] Tuya, J., Suárez-Cabal, M.J., de la Riva, C. 2007. Mutating database queries. Inform. Software Tech. 49, 4, 398-417.

[17] Tuya, J., Suárez-Cabal, M.J., de la Riva, C. 2010. Full predicate coverage for testing SQL database queries. Softw. Test. Verif. Rel, in press.

[18] Veanes, M. Grigorenko, P. de Halleux, P., Nikolai, T. 2009.Symbolic Query Exploration. In Proceedings of the 11th International Conference on Formal Engineering Methods, LNCS, Vol. 5885, Springer, 49-68.

[19] Willmor, D., Embury S.M. 2006. An intensional approach to the specification of test cases for database applications. In Proceedings of the 28th International Conference on Software Engineering. ACM New York, NY, 102-111.

[20] Zhang, J., Xu, C., Cheung, S.C. 2001. Automatic generation of database instances for white-box testing. In Proceedings of the 25th International Computer Software and Applications Conference. IEEE Computer Society, Washington, DC, 161-165.

[21] Zhou, C., Frankl, P. 2009. Mutation testing for Java database applications. In Proceedings of the 2nd International Conference on Software Testing Verification and Validation. IEEE Computer Society, Washington DC, 396-405.

[22] The Alloy Analyzer. http://alloy.mit.edu/

[23] SQLFpc tool. http://in2test.lsi.uniovi.es/sqlfpc/

[24] SQLMutation tool http://in2test.lsi.uniovi.es/sqlmutation/