Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

Phil McMinn
Department of Computer Science
University of Sheffield

Chris J. Wright
Department of Computer Science
University of Sheffield

***Abstract*—There has been much attention to testing applications that interact with database management systems, and the testing of individual database management systems themselves. However, there has been very little work devoted to testing arguably the most important artefact involving an application supported by a relational database – the underlying schema. This paper introduces a search-based technique for generating database table data with the intention of exercising the integrity constraints placed on table columns. The development of a schema is a process open to flaws like any stage of application development. Its cornerstone nature to an application means that defects need to be found early in order to prevent knock-on effects to other parts of a project and the spiralling bug-fixing costs that may be incurred. Examples of such flaws include incomplete primary keys, incorrect foreign keys, and omissions of `NOT NULL` declarations. Using mutation analysis, this paper presents an empirical study evaluating the effectiveness of our proposed technique and comparing it against a popular tool for generating table data, *DBMonster*. With competitive or faster data generation times, our method outperforms *DBMonster* in terms of both constraint coverage and mutation score.**

I. Introduction

A database schema specifies the types of data that will be used by an application, how the data will be organized into tables, which data values are valid, and what relationships may exist between them. For any application backed by a database, the inception of the schema is one of the first stages of the development cycle. It is perhaps surprising that there has been almost no work devoted to schema testing. Previous research has instead concentrated on the interface between the database and application code [1], [2], [3], or on testing the underlying database or management system [4], [5], [6].

A key aspect of a database schema is the definition of its *integrity constraints*. Integrity constraints specify what data values are valid for each column of a table – for example specifying that they should be not null or in a particular range denoted by a "check" constraint. They also specify how types of data are interconnected through foreign key constraints, and which values are responsible for identifying sets of other values through primary key declarations. The definition of correct integrity constraints over a schema is a key concern for testing and is addressed in this paper through a search-based approach. Search-based methods have been successfully applied to many other forms of test data generation [7] – most notably structural testing – but have not, until now, been applied to generating database table data. The approach creates a series of SQL `INSERT` statements containing data generated to exercise each integrity constraint of a schema as true or false. The aim of the technique is to highlight potential flaws in the design of the schema through the generation of data that is allowed into the database where it should have been rejected by the database management system (DBMS), and vice versa.

The paper evaluates the search-based technique with an empirical study that required the design of a series of mutation operators. These operators mutate the integrity constraints of a schema and give an indication of the thoroughness of the schema's test suite. The search-based technique, implemented into a tool called *SchemaAnalyst*, is compared with the publicly available open-source tool *DBMonster*, which uses a random method for generating database table data. Employing 25 case studies and three DBMSs – Postgres, HSQLDB, and SQLite – the experiments evaluate the algorithms implemented in both of the tools.

The study finds that, for all of the case studies, *SchemaAnalyst* obtains higher constraint coverage than *DBMonster* while reaching 100% coverage on two schemas for which *DBMonster* covers less than 10% of the constraints. The experiments also reveal that *SchemaAnalyst*'s mutation score is higher than the random method's for 96% of the schemas and that it succeeds in generating data for six schemas that cause the established method to crash. *SchemaAnalyst* achieves these results with generated data sets that are substantially smaller than *DBMonster*'s and in an amount of execution time that is competitive or faster.

The contributions of this paper are therefore as follows:

1) A search-based method for generating data that is capable of satisfying and negating the integrity constraints of schemas across multiple DBMSs (Section III).
2) A set of mutation operators for evaluating table data intended to test integrity constraints (Section IV).
3) An empirical study with 25 schemas and 3 different DBMSs that compares the efficacy and efficiency of our search-based technique to the same measures for *DBMonster*, a popular publicly available and open-source tool for generating table data (Section V).

II. The Need for Schema Testing

Figure 1 depicts an SQL schema declaration for the database of a flight booking application that involves two tables. The first declaration, for a table called `Flights`, defines information to be stored in the database about actual flights, with the declaration of eight columns and their types (strings, integers, characters, and times). The second declaration, for the `FlightAvailable` table, defines seat availability information. Both tables have *integrity constraints* declared on them.

Integrity constraints protect the consistency of data in a relational database by causing the database management system to reject the insertion of table rows – via SQL `INSERT` statements – that do not satisfy certain restrictions. `NOT NULL` constraints ensure that values for a table column are never `NULL`. `PRIMARY KEY` declarations define subsets of columns that must be unique for each row, so that complete rows of data can be subsequently identified and retrieved (columns for which row values must be unique may also be declared with a `UNIQUE` constraint). `FOREIGN KEY` constraints link rows in one table to rows in another – for example, rows in `FlightAvailable` are linked to those in `Flights` through the `FLIGHT_ID` and `SEGMENT_NUMBER` columns. That is, every row in `FlightAvailable` must have a value pair for `FLIGHT_ID` and `SEGMENT_NUMBER` that appear as primary key values for the `Flights` table. Finally, `CHECK` constraints involve the declaration of arbitrary predicates over table data. In Figure 1, the `Flights` table has a `CHECK` constraint on the `MEAL` column, indicating table cells of that column must be of a certain character. A `CHECK` constraint may also involve relational predicates.

The definition of the structure of the schema is one of the first steps in developing an application supported by a database. As such, any errors made in this stage can be costly to rectify later in the development cycle. For example, suppose that the column `SEGMENT_NUMBER` (denoting an individual stage of a long-haul journey) was omitted from the primary key for the `Flights` table, with rows uniquely identified by `FLIGHT_ID` only. Now, flight data involving more than one flight stage may be rejected, due to the violation of the table's primary key. Such defects can be caught early in the definition of the schema through testing. Test data needs to be generated that first inserts some arbitrary data into the database. For instance, consider

```
INSERT INTO Flights VALUES('a', 1, ... )
```

followed by an `INSERT` repeating the same `FLIGHT_ID`:

```
INSERT INTO Flights VALUES('a', 2, ...)
```

The latter statement is rejected by the DBMS when it should not have been, thus pointing out the mistake in the schema to the tester. Therefore, part of the schema testing problem is the generation of test data suitable for identifying defects in the schema. Although previously overlooked in the literature, this is the challenge addressed by this paper.

```
CREATE TABLE Flights(
  FLIGHT_ID         CHAR(6) NOT NULL,
  SEGMENT_NUMBER    INT     NOT NULL,
  ORIGINAL_AIRPORT CHAR(3),
  DEPART_TIME       TIME,
  DEST_AIRPORT      CHAR(3),
  ARRIVE_TIME       TIME,
  MEAL              CHAR(1),
  PRIMARY KEY(FLIGHT_ID, SEGMENT_NUMBER),
  CHECK(MEAL IN ('B', 'L', 'D', 'S'))
);

CREATE TABLE FlightAvailable (
  FLIGHT_ID               CHAR(6) NOT NULL,
  SEGMENT_NUMBER          INT     NOT NULL,
  FLIGHT_DATE             DATE    NOT NULL,
  ECONOMY_SEATS_TAKEN     INT,
  BUSINESS_SEATS_TAKEN    INT,
  FIRSTCLASS_SEATS_TAKEN INT,
  PRIMARY KEY(FLIGHT_ID, SEGMENT_NUMBER),
  FOREIGN KEY(FLIGHT_ID, SEGMENT_NUMBER)
    REFERENCES Flights(FLIGHT_ID, SEGMENT_NUMBER)
);
```

Figure 1. An Example Schema for a Flight Booking Application (integrity constraints are highlighted in bold).

Previous work on database testing has only sought to test database interaction from within an application (e.g., [1], [2], [3]) or testing either the database or the DBMS (e.g., [4], [5], [6], [8], [9], [10]). During application testing, the implicit assumption is that the database schema is correct. However, since integrity constraints encode a logic of their own, it is also important to test the schema itself, prior to integration with an application. The next section introduces our technique for schema testing through the generation of data to exercise integrity constraints in databases.

III. Search-Based Generation of Database Table Data

Due to the potential presence of arbitrary `CHECK` constraints, the generation of database table data to test a database schema is non-trivial in the general case. This section describes our search-based technique for generating table data, implemented in a tool called *SchemaAnalyst*, as shown in Figure 2. *SchemaAnalyst* supports three DBMSs – Postgres, HSQLDB, and SQLite – and further provides the schema mutation analysis methods described in Section IV.

A. Abstract Representation of Schemas

The first step of our technique involves converting a database schema to a DBMS-independent form known as its "abstract representation." Different DBMSs support different sets of column types, and as part of this conversion process, a mapping must be made from each concrete DBMS column type to one of seven "universal" types: *Boolean*, *DateTime*, *Date*, *Numeric*, *String*, *Time*, and *Timestamp*. The mapping of several common DBMS column types to these universal types is shown in Table I. Each universal type belongs to one of two general categories. *Atomic* types are singular values, such as the *Numeric* and *Boolean* universal types. *Compound* types are made up of a number of values of atomic type, and include Strings, which consist of a series of *Numeric* values corresponding to the String's individual characters.

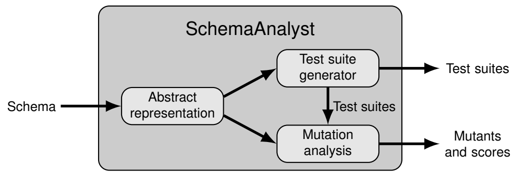


Figure 2. Overview of the Main Components of *SchemaAnalyst*.

B. Overview of the Test Data Generation Algorithm

The goal of the data generation algorithm is to generate a test suite consisting of `INSERT` statements containing data values that exercise each of the schema's integrity constraints as true and false. That is, for each integrity constraint, the test suite will contain at least one `INSERT` statement for which data will be accepted into the table where the constraint is satisfied, and one for which the data will be rejected because the constraint is violated.

The percentage of constraints satisfied and violated by a test suite is referred to as the level of *constraint coverage*. Since each `INSERT` statement has the potential to be rejected, an important part of the test suite generation algorithm is to keep track of both the `INSERT` statements whose data is successfully entered into the database and the current database "state." This is because the acceptance or rejection of data involved in each `INSERT` statement is dependent on the data entered into the database by the `INSERT` statements that went before it. For example, a `PRIMARY KEY` constraint cannot be violated unless the column values are generated that match those of an existing row in the database.

The test suite generation algorithm requires an initially empty database state, and proceeds as follows:

Stage 1) Satisfy the schema: For each schema table, generate n_s ($n_s \geq 2$) rows of data, with the aim of satisfying all integrity constraints in the schema. To prevent `PRIMARY KEY` and `UNIQUE` constraints from being trivially satisfied n_s should not be less than two. Furthermore, `FOREIGN KEY`, `UNIQUE`, and `CHECK` constraints can be trivially satisfied with `NULL` values, so the usage of `NULL` values is disallowed for this stage. Assuming successful generation of data that satisfies all of the integrity constraints, `INSERT` statements are formed for each row of table data generated. These `INSERT`s will be accepted by the DBMS, the data inserted becomes the state of the database, and 50% coverage will have been obtained.

Stage 2) Individually violate each integrity constraint: For each constraint c in each table t of the schema, generate n_v ($n_v \geq 1$) rows of data for t and any tables referenced through `FOREIGN KEY` declarations, with the aim of violating c while satisfying all other constraints. Assuming successful data generation, `INSERT` statements may be formed for each table row of data. The `INSERT` statement for t corresponding to the violation of c will be rejected by the DBMS, while rows for tables referenced by t will be inserted and entered into the state of the relational database.

Table I
MAPPINGS OF DBMS COLUMN TYPES TO UNIVERSAL TYPES

DBMS Type	**Universal Type**	**General Type**	**Default Value**
`BOOLEAN`	Boolean	Atomic	False
`DATETIME`	DateTime	Compound	0/0/0 00:00:00
`DATE`	Date	Compound	0/0/0
`DECIMAL`, `DOUBLE`, `FLOAT`, `INTEGER`, `NUMERIC`, `REAL` ...	Numeric	Atomic	0
`CHAR`, `VARCHAR` ...	String	Compound	Empty string
`TIMESTAMP`	Timestamp	Atomic	0
`TIME`	Time	Compound	00:00:00

C. Formation of the Fitness Function

For both stages of the aforementioned overall algorithm, a constraint system is constructed. For stage 1, the constraint system is simply the conjunction of all integrity constraints. For stage 2, the constraint system is the conjunction of all constraints to be satisfied and the negation of the constraint to be violated. From this constraint system, a *fitness function* is formed. The purpose of a fitness function in search-based testing is to guide a search algorithm towards some test goal of interest. Here, the test goal is the generation of rows of `INSERT` statement data that will satisfy a constraint system formed for stage 1 or 2. The fitness function computes a numeric value to be minimized by the search. That is, the closer the generated row values are to satisfying the constraint system, the lower the fitness value will be. When the fitness function evaluates to zero, the generated row values satisfy the constraint system.

The fitness function involves the computation of a series of *distance values*. A distance value indicates how close some constraint, or some component of a constraint, was to being satisfied. In fitness function design, it is important to balance different objectives such that one aspect of the search problem does not dominate the rest. In order to ensure that all constraints are weighted equally, their distance values are normalized to the range [0,1] using the function $norm(d) = \frac{d}{d+1}$ (due to Arcuri [11]). The normalized distance values for each of the constraints are added together to form the complete fitness value. We now detail how distance values are computed for each type of integrity constraint using the functions shown in Figure 3.

1) Primary Keys: Distance value computation for primary key constraints involves the use of the row_unique and row_non_unique functions, shown in Figure 3. These functions compare a row r, a tuple of column values, to a series of other rows $s_1...s_n$, which are further tuples of column values. For primary key distance computation, the column values for r are the primary key column values for the row of data currently being generated for an `INSERT` statement, while $s_1...s_n$ consist of primary key column values for rows already inserted into the database state or are currently being generated at the same time as r. If the primary key is to be

value_dist(a, op, b)

a	b	
NULL	*any*	return 1
any	NULL	return 1
Atomic	Atomic	return $norm(atomic\_dist(a, op, b))$
Compound	Compound	return $norm(compound\_dist(a, op, b))$

atomic_dist(a, op, b)

op	
$=$	if $(\lvert a - b\rvert = 0)$ then return 0 else return $\lvert a - b\rvert + 1$
$\neq$	if $(\lvert a - b\rvert \neq 0)$ then return 0 else return 1
$<$	if $(a - b < 0)$ then return 0 else return $(a - b) + 1$
$\leq$	if $(a - b \leq 0)$ then return 0 else return $(a - b) + 1$
$>$	if $(b - a < 0)$ then return 0 else return $(b - a) + 1$
$\geq$	if $(b - a \leq 0)$ then return 0 else return $(b - a) + 1$

compound_dist$((a_1 \dots a_p), op, (b_1 \dots b_q))$

op	
$=$	return $\lvert p - q\rvert + \sum_{i=1}^{min(p,q)} norm(atomic\_dist(a_i, =, b_i))$

$\neq$	if $(p \neq q)$ then return 0
	else return $min_{i=1}^{min(p,q)} norm(atomic\_dist(a_i, \neq, b_i))$

```
other   $d \leftarrow 0$
  while $(i \leq min(p, q) \wedge d = 0)$
      if $(a_i \neq b_i)$ then $d \leftarrow norm(atomic\_dist(a_i, op, b_i))$
      $i \leftarrow i + 1$
  end while
  return $d + atomic\_dist(p, op, q)$
```

and_dist$(d_1, \dots, d_n)$
return $norm(\sum_{i=0}^{i=n} d_i)$

or_dist$(d_1, \dots, d_n)$
return $min_{i=0}^{i=n} d_i$

row_dist$((a_1 \dots a_n), op, (b_1 \dots b_n))$
return $norm(\sum_{i=1}^{i=n} value\_dist(a_i, op, b_i))$

row_unique$(r, s_1 \dots s_n, allow\_null))$
if $(allow\_null \wedge$ any values of $r =$ NULL$)$ then return 0
else return $norm(\sum_{i=1}^{i=n} row\_dist(r, \neq, s_i))$

row_non_unique$(r, s_1 \dots s_n, allow\_null))$
if $(allow\_null \wedge$ any values of $r =$ NULL$)$ then return 0
else return $min_{i=1}^{i=n} row\_dist(r, =, s_i)$

Figure 3. Distance Functions.

satisfied, row_unique is used, whereas row_non_unique is used if the constraint should be violated. (The $allow\_null$ parameter results in the function trivially returning true if any values in r are NULL, however this is disabled for primary key constraint evaluation). Both functions involve the collation of distances for each row using the row_dist function. The row_dist function computes how far some relational predicate involving two rows was to being satisfied, performed by summing individual distances for pairs of column values from the $value\_dist$ function.

Take the example of Figure 4, where rows 1 and 2 have already been entered into the database for the Flights table of Figure 1. A new row (row 3) is required that violates the primary key constraint. That is, values are required for FLIGHT_ID and SEGMENT_NUMBER that match

	FLIGHT_ID	SEGMENT_NUMBER	...
1	'UA21'	1	...
2	'UA3750'	1	...
3	'UA22'	2	...

Figure 4. Example for Demonstrating How Fitness Values Are Calculated.

the values already in the database for those columns in either row 1 or 2. To compute the distance, row_non_unique is called with r = ('UA22', 2) and s_1 = ('UA21', 1), s_2 = ('UA3750', 1). This function calls row_dist with the row pairs (r, s_1) and (r, s_2), respectively, with '=' for the op argument, since equal column values are required for constraint violation. Then, row_dist compares the column values for each pair of rows using the $value\_dist$ function. The $value\_distance$ of 'UA22' and 'UA21' is computed by a call to $compound\_dist$, since the column is of type CHAR, which is mapped to the *String* universal type, a compound of *Numeric* values (see Table I). 'UA22' and 'UA21' are identical but for the last character, so the distance is $norm(1 + 1)$. Following this, the $value\_distance$ function is called for the SEGMENT_NUMBER column, with the values 2 and 1. Integer values are *Numeric* "atomic" types, so $atomic\_dist$ is called, evaluating to $1 + 1$. Therefore, the overall result of row_dist(('UA22', 2),'=',('UA21', 1)) is $norm(norm(norm(1 + 1)) + norm(1 + 1))$ = 0.516. The corresponding calculation comparing row 2 and 3 is row_dist(('UA22', 2),'=',('UA3750', 1)) = $norm(norm(norm(1 + 1) + (2 + 1)) + norm(1 + 1))$ = 0.592. Since $0.516 < 0.592$, the individual row distance calculations indicate that row 3 is "closer" to being the same to row 1 than row 2. Therefore, the overall value returned by row_non_unique is 0.516.

Note that row 3 satisfies the primary key constraint. Performing the sub-fitness calculation using row_unique, which performs row distance calculations using the $\neq$ operator rather than $=$, returns a value of 0.

2) Unique constraints: Distance values for satisfying and falsifying UNIQUE constraints are computed in the same way as for primary keys. UNIQUE constraints can be satisfied with the use of NULL values and so $allow\_null$ is true for calls to row_unique in stage 2 of the overall algorithm.

3) Foreign Keys: Foreign key distance value computation is performed using the row distance functions of Figure 3 again, but with a different configuration. For violating foreign keys, unique values of r are required that do not match corresponding column values in rows of the referenced table. Therefore, row_unique is used, where $s_1 \dots s_n$ are tuples of foreign key column values in each row of the referenced table. For satisfying foreign key constraints, values of r are required that match the values in corresponding columns of rows in the referenced table, so row_non_unique is used. Foreign keys may be satisfied with NULL values – that is, in stage 2 of the overall algorithm, $allow\_null$ is true for calls to the row_non_unique function.

4) Not Null constraints: Generating a distance value for a NOT NULL constraint involves checking the column value that needs to be NULL (violating the constraint) or not NULL (satisfying). Where the intended value is found, the distance is 0, otherwise it is 1.

5) Check constraints: There are three different types of CHECK constraint. The first type is composed of arbitrary relational predicates, for example CHECK DAY >= 1 AND DAY <= 31. For such predicates, distance values are collated for each sub-predicate using the $value\_distance$ function. In this instance, satisfaction of the constraint involves making two calls: $value\_distance(\texttt{DAY}, \geq, 1)$ and $value\_distance(\texttt{DAY}, \leq, 31)$ An overall distance value is formed depending on the conjunction and disjunction of the sub-predicates using and_dist or or_dist, respectively. Falsifying the constraint involves negation of the overall predicate and following the same procedure – that is, computing a distance for DAY < 1 OR DAY > 31.

The second type of CHECK constraint uses the BETWEEN operator to ensure that a value falls into a certain range – for example, CHECK MONTH BETWEEN 1 AND 12. Such constraints are rewritten internally to the equivalent relational predicate (e.g., CHECK MONTH >= 1 AND MONTH <= 12) and the distance value computed as previously discussed.

The third type of CHECK constraint involves the use of the IN operator, as in the example of Figure 1 with the MEAL field. The IN operator checks that a column value takes on one of the values specified in a set. It is equivalent to writing a disjunction of equality predicates, that is, MEAL = 'B' OR MEAL = 'L' OR MEAL = 'D' OR MEAL = 'S'. Predicates are internally rewritten to this form, and distance calculation follows that for arbitrary relational predicates, as detailed previously.

D. Search Using the Alternating Variable Method

Our technique applies a specialized version of Korel's *Alternating Variable Method* [12] (AVM) as the search technique to minimize the fitness function. The rows of values for each INSERT statement are formed into a list with each value set to its default value, as summarized in Table I. The AVM sequentially makes adjustments to each value in the list, referred to as "moves." After each move, the list of values is evaluated according to the fitness function. If a move on a value leads to an improvement in fitness, the new adjusted value is kept, else the value reverts to the original.

The initial set of moves attempted for a value are referred to as "exploratory" moves. The value first has its NULL status flipped. If, following this move, the value is not NULL, further moves are performed depending on the value's universal type. If the value is a *Boolean*, its value is flipped. If the value is of *Numeric* or *Timestamp* type, two moves are attempted, one which decreases the value, and one that increases the value. If either move is found to improve fitness, a series of "pattern" moves are made, which accelerate modifications to the value in the direction of improvement. Pattern move steps are computed using the function $m_i = 2^i \cdot 10^{-d} \cdot dir$, where m_i is the i^{th} successive move, dir is the direction of improvement, $dir \in \{-1, 1\}$, and d is the number of decimal places involved in the numeric value (as originally described in [13]). Pattern moves continue until a move is made that no longer improves fitness. Finally, compound values are simply treated as sub-lists of the overall list of values, with the AVM sequentially performing moves (bar the NULL move) as for the overall tuple of data values. For Strings, additional exploratory moves may be performed on the whole String, involving the addition or removal of characters.

The AVM terminates when either a fitness of zero has been reached, indicating that the required test data has been found, or when a complete cycle of exploratory moves has been made through the list without any improvement in fitness. If the latter occurs, the AVM is restarted but with random, rather than default values, for each column. This process continues until test data is found or there has been 100,000 evaluations of the fitness function.

IV. Mutation Analysis of Schemas

In order to assess the effectiveness of test suites generated by our technique, we devised mutation operators for each type of integrity constraint. Mutants are produced as follows:

1) Primary Keys: Primary key mutants take three forms: *Add Column*, *Replace Column*, and *Remove Column*. Mutants are produced by iterating through a table's columns. If the column is a part of the original table's primary key, a mutant is produced with that column removed from the primary key. If the column is not a part of the original table's primary key, a mutant is produced with the column added, along with the creation of further mutants where the column replaces each existing primary key column in turn. For the Flights table of Figure 1, an example of a remove column mutant would be one with the primary key declaration PRIMARY KEY(FLIGHT_ID) (i.e., with SEGMENT_NUMBER missing) while an example of an add column mutant includes PRIMARY KEY(FLIGHT_ID, SEGMENT_NUMBER, ORIGINAL_AIRPORT). An example of a column replacement includes PRIMARY KEY(SEGMENT_NUMBER, ORIGINAL_AIRPORT), where FLIGHT_ID is replaced with ORIGINAL_AIRPORT. This approach produces a total of 31 primary key mutants for the flight booking example.

2) Unique constraints: Mutant production for UNIQUE constraints works in a similar fashion to primary keys, except that, while a table can have several UNIQUE declarations, it can only have one primary key. The set of UNIQUE constraints is collated, along with an additional "empty" constraint with no columns. For each constraint in the set, the algorithm for adding and removing columns is followed as for primary keys, taking care to avoid the production of identical mutants from the mutation of different unique constraints within the same table. This results in 13 total mutants for the flight booking example.

3) NOT NULL constraints: The `NOT NULL` mutation operator iterates through the non-primary key columns of each table of a schema and reverses its `NOT NULL` status. That is, if a column is declared to be `NOT NULL`, a mutant is produced with the `NOT NULL` constraint removed. If a column if found to not have a `NOT NULL` constraint, a mutant is produced with one added. For instance, nine `NOT NULL` mutants are produced for the flight booking example.

4) Foreign Keys: The production of mutants for `FOREIGN KEY` constraints involves iterating through the foreign keys and producing a mutant where each foreign key column is removed from the schema. For the flight booking example, this results in two foreign key mutants being produced.

5) Check constraints: Finally, the production of `CHECK` constraint mutants simply involves iterating through the `CHECK` constraints of a schema and generating mutants by each constraint one at a time. For example, one mutant would be produced from the `Flights` table of Figure 1, with the `CHECK` constraint on the `MEAL` column removed.

This gives a total of 56 mutants created for the flights example. Some mutants generated according to the aforementioned rules are rejected as invalid schema by certain DBMSs. With the flight booking example of Figure 1, Postgres will not allow columns to be added or removed from the primary key in the `Flights` table. This is because of the foreign key reference from the `FlightAvailable` table, which requires the `FLIGHT_ID` and `SEGMENT_NUMBER` column pair to uniquely index rows in the `Flights` table. In Mutation Analysis, "illegal" mutants of this nature are referred to as *still-born* [14]. However, the same mutants may be perfectly acceptable to other DBMSs – for example SQLite – and as such not completely without worth. Therefore, we refer to such mutants as *quasi-mutants* and we study their frequency and characteristics in Section V.

V. Empirical Study

A. Case Studies

Following Houkjær et al.'s observation that complex relational schemas often exhibit features like composite keys and multi-column foreign-key relationships [15], we selected the 25 schemas in Table II as case studies. While several of these schemas originate from the Postgres samples available from the `PgFoundry.org` site (e.g., BookTown, DellStore, FrenchTowns, Iso3166, and Usda), others were used in previous studies of database-aware testing techniques (e.g., the NIST schemas [8], RiskIt and UnixUsage [3], and JWhoisServer [16]). Even though we derived several schemas (e.g., CoffeeOrders, CustomerOrder, Person, and Products) from examples found in textbooks, laboratory assignments, and online tutorials, it is important to note that many are complex and, as the results in Section V-D reveal, difficult for the well-established *DBMonster* to handle efficiently. Moreover, we balanced the use of example schemas by incorporating Cloc, JWhoisServer, RiskIt, and UnixUsage – all schemas from real-world database applications.

Table II
Case study schemas used in the empirical study

Schema	*Tables*	*Columns*	*Checks*	*Foreign keys*	*Not Nulls*	*Primary keys*	*Uniques*	*Total Constraints*
BankAccount	2	9	0	1	5	2	0	8
BookTown	23	69	1	0	17	11	0	29
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	9	5	0	18
CustomerOrder	7	32	1	7	27	7	0	42
DellStore	8	52	0	0	36	0	0	36
Employee	1	7	3	0	0	1	0	4
Examination	2	21	6	1	0	2	0	9
Flights	2	13	1	1	6	2	0	10
FrenchTowns	3	14	0	2	13	0	8	23
Inventory	1	4	0	0	0	1	1	2
Iso3166	1	3	0	0	2	1	0	3
JWhoisServer	6	49	0	0	44	6	0	50
NistDML181	2	7	0	1	0	1	0	2
NistDML182	2	32	0	1	0	1	0	2
NistDML183	2	6	0	1	0	0	1	2
NistWeather	2	9	5	0	2	2	0	9
NistXTS748	1	3	1	0	1	0	1	3
NistXTS749	2	7	1	1	3	2	0	7
Person	1	5	1	0	5	1	0	7
Products	3	9	4	2	5	3	0	14
RiskIt	13	56	0	10	15	11	0	36
StudentResidence	2	6	3	1	2	2	0	8
UnixUsage	8	32	0	7	9	7	0	23
Usda	10	67	0	0	30	0	0	30
Total	111	542	27	40	231	68	11	377

We also configured each of the schemas to run on three representative database management systems: Postgres, HSQLDB, and SQLite. We picked these DBMSs because, beyond being widely used in many real-world database applications, they all support the expression and enforcement of constraints involving primary and foreign keys, and `UNIQUE`, `NOT NULL`, and arbitrary `CHECK` constraints. While these DBMSs share many common features (e.g., support for the latest SQL standards), they also vary in the auxiliary commands that they provide and their overall architecture – Postgres runs in a stand-alone server while SQLite and HSQLDB are in-memory DBMSs respectively implemented in the C and Java languages. Finally, we configured the data generation methods to interact with the DBMSs through standard Java database connectivity (JDBC) drivers.

B. Technique Configuration

Beyond allowing at most 100,000 fitness evaluations, we configured the AVM with $n_s = 2$ and $n_v = 1$, thus aiming to satisfy the database's schema with two rows per table and violating each of the schema's constraints with one row. Using *DBMonster*'s standard configuration interface, we permitted the tool to perform no more than 100,000 data generation attempts. We also required *DBMonster* to generate at least 50 rows per table since a preliminary investigation revealed that fewer rows than this resulted in the tool not being able to reliably kill any mutants. For all of the data types represented in the chosen schemas, we configured *DBMonster* to use its default data generators. Since *DBMonster* is a third-party tool that does not record the database interactions that occur during data generation, we used Cobb et al.'s approach [16] to capture these SQL statements. Due to the fact that *DBMonster* frequently crashed when generating data for schemas hosted by SQLite and HSQLDB, we only ran this tool with Postgres. In contrast, we used the AVM to correctly generate data for all three of the DBMSs. Since the AVM and *DBMonster* both incorporate randomness, we applied each technique to every schema for 30 trials, thus controlling threats to validity.

Both *SchemaAnalyst* and *DBMonster* are written in the Java language and respectively compiled and executed with version 1.7 of the compiler and Java virtual machine. During the empirical study, the methods were executed on Ubuntu 12.04 workstations equipped with a quad-core 3.3 GHz CPU and configured to use the 3.2.0-30 GNU/Linux 64-bit kernel. Both the data generators and the DBMSs were stored and executed on a 230 GB local disk.

C. Evaluation Metrics

To evaluate efficiency, we measured the execution time of the AVM and *DBMonster*, including all of the fitness function evaluations and database interactions that take place during data generation. As one assessment of effectiveness, we calculated constraint coverage – the ratio of constraints satisfied and violated to the total number of schema constraints multiplied by two. Furthermore, we employ mutation analysis, as described in Section IV, as an additional effectiveness measure. For a generated set of database interactions D, if Q denotes the set of quasi-mutants and K and N respectively stand for the sets of killed and not-killed mutants, then $M_D = |K \cup Q| / |K \cup N|$ computes the higher-is-better mutation score. That is, we declare a mutant to be dead when either the generated data in D kills it or it is "quasi" and thus rejected for not conforming to DBMS-specific requirements. To calculate M_D, the mutation analysis process iteratively applies each operator to a schema S, repeatedly producing a mutant S' against which each operation in D is executed. We say that a $d \in D$ kills S' when it produces a different output than it did when executed with relational schema S.

Since the mutation scores of *DBMonster* varied across the 30 trials, Figure 6 uses a box-and-whisker plot where the center circle represents the median and the box itself shows the distribution of data between the upper and lower quartiles, thus including 50% of the data. Exclusive of the outlying values that are depicted with open circles, the upper and lower whiskers stand for the minimum and maximum value, respectively. It is important to note that, across the 30 trials, AVM generated data that reliably killed the same mutants and thus resulted in an unchanged mutation score. As such, Figure 6 uses a bar chart to visualize these values. Similarly, since the number of generated database interactions, or $|D|$, did not vary for AVM but was different across trials for *DBMonster*, Figure 6 visualizes this data in the same way as was chosen for the mutation score.

Table III
CONSTRAINT COVERAGE

Schema	AVM (%)	*DBMonster* (%)
BankAccount	100.0	56.3
BookTown	100.0	51.7
Cloc	*(no constraints defined)*	
CoffeeOrders	100.0	50.0
CustomerOrder	100.0	9.5
DellStore	100.0	50.0
Employee	100.0	55.0
Examination	100.0	72.2
Flights	100.0	70.0
FrenchTowns	100.0	70.0
Inventory	100.0	75.0
Iso3166	100.0	50.0
JWhoisServer	100.0	50.0
NistDML181	100.0	75.0
NistDML182	100.0	50.0
NistDML183	100.0	100.0
NistXTS748	100.0	72.2
NistXTS749	100.0	21.4
NistWeather	100.0	68.7
Person	100.0	50.0
RiskIt	100.0	4.1
Products	96.4	59.3
StudentResidence	100.0	62.5
UnixUsage	97.8	59.3
Usda	100.0	50.0

In addition to relying on the data visualizations, we compute Tukey's five-number descriptive statistics [17] for a chosen evaluation metric when comparing *DBMonster* and *SchemaAnalyst* across all of the 25 schemas. Reported in the format (minimum, lower quartile, median, upper quartile, and maximum), these values enable us to make observations about the strengths and weaknesses of the two techniques when they are applied to many different types of schemas.

D. Results Analysis

1) Quasi Mutants: Figure 5 shows the number of mutants that are and are not classified as quasi-mutants. This bar chart reveals that, for some schemas like Cloc, DellStore, NistDML183, and Usda, the mutation analysis process does not produce any quasi-mutants for any of the DBMSs. For other schemas like BookTown, Employee, French-Towns, Inventory, Iso3166, JWhoisServer, NistWeather, and NistXTS748 the mutation operators only produce a small number of quasi-mutants for certain DBMS(s). This figure highlights the fact that, in contrast to HSQLDB and Postgres, the most permissive DBMS, SQLite, never produces any quasi-mutants – further reinforcement of the need for a multi-DBMS approach. It is also important to note that quasi-mutants do not dominate the total number of mutants for any of the schemas. A low number of quasi-mutants means that Section V-D3's mutation scores are not artificially inflated and thus good indicators of effectiveness.

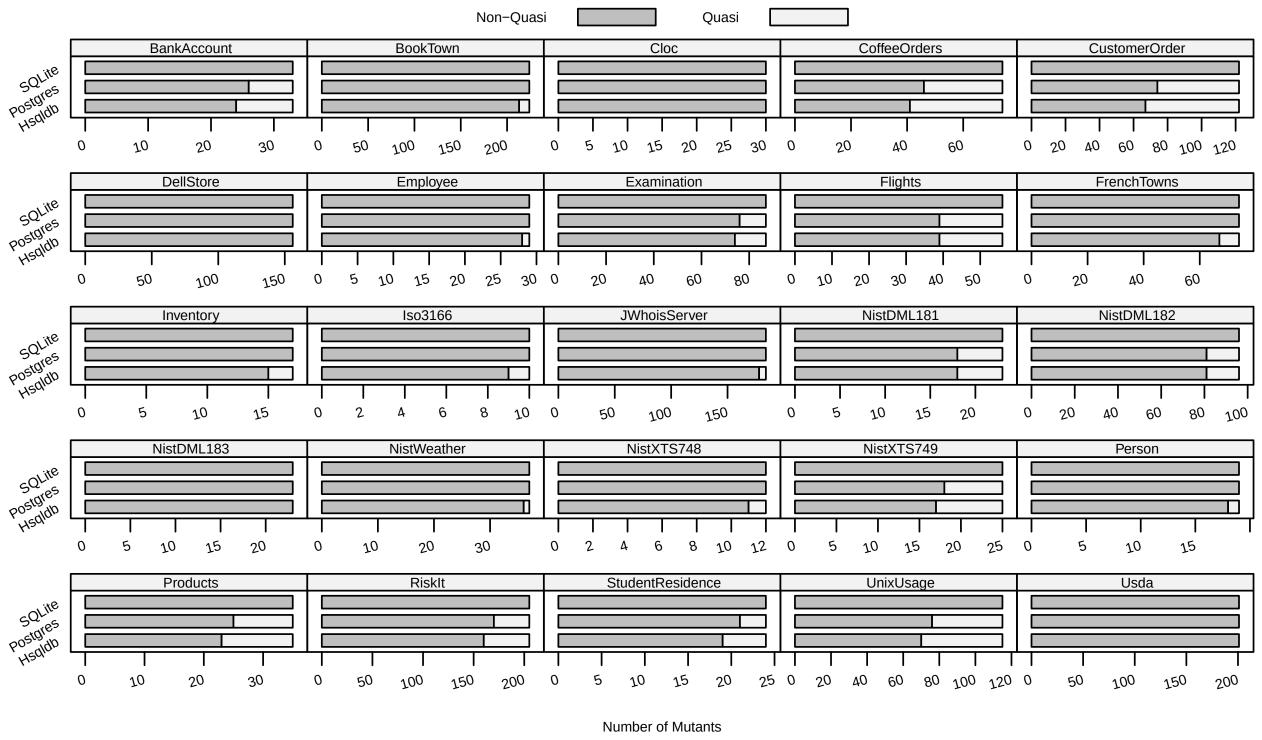


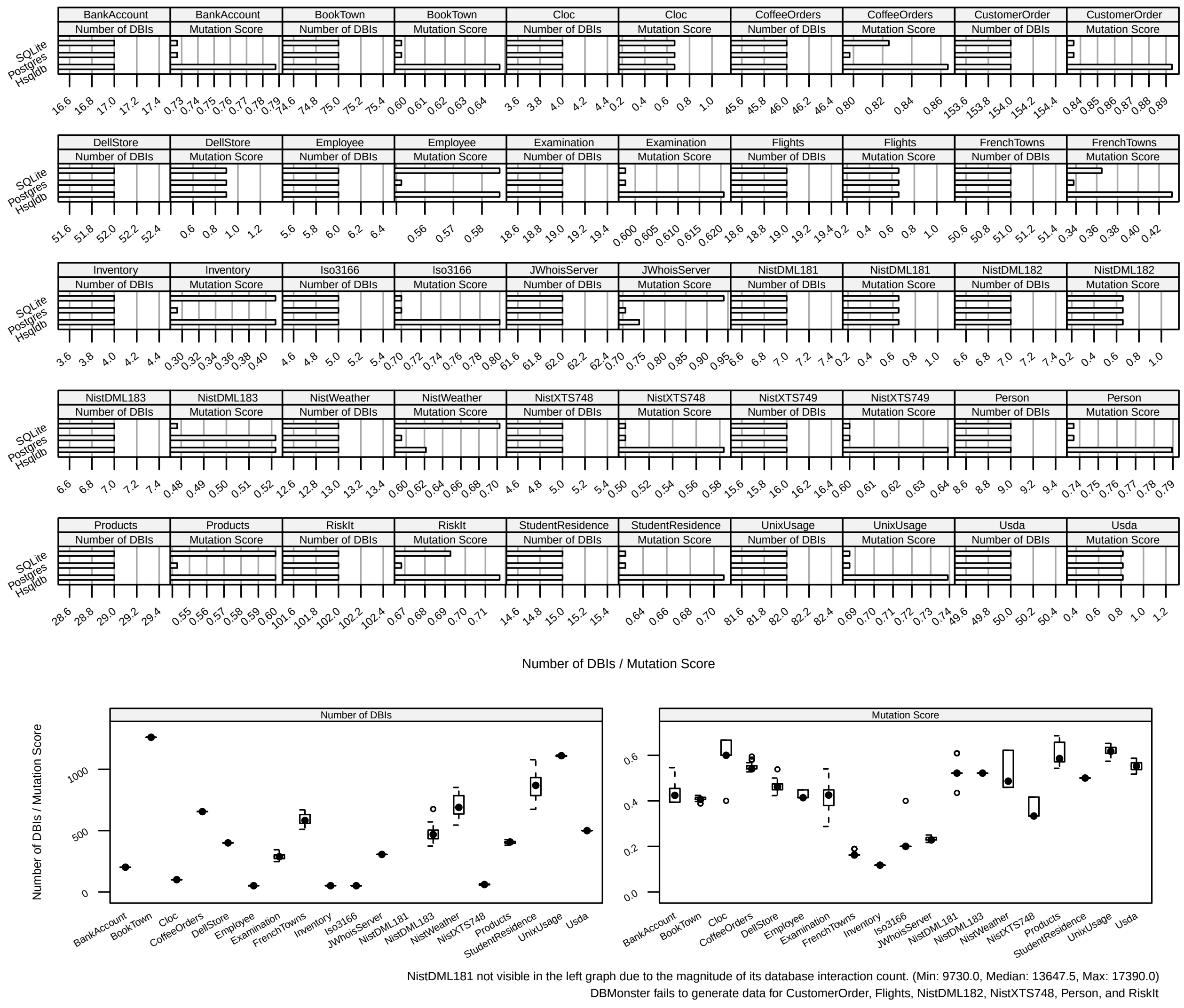
Figure 5. The Number of Non-Quasi-Mutants and Quasi-Mutants for All of the Schemas.

2) Constraint Coverage: Although not the primary focus of this section, we briefly report on both method's constraint coverage because data that does not cover a constraint will not be able to kill the mutants associated with it. Table III shows that, across the 30 trials, the AVM achieves 100% constraint coverage for all schemas, with the exception of Products and UnixUsage, for which it attains 96.43% and 97.83% constraint coverage, respectively. In contrast, *DBMonster*'s constraint coverage scores are often substantially lower: while this method covers 72% of Examination's constraints, it only reaches 50% coverage for schemas like DellStore and Iso3166, and is only able to cover 4.1% of the constraints in RiskIt. These results indicate that, on average across all of the schemas, *DBMonster* covers significantly fewer constraints than *SchemaAnalyst* and is thus less likely to kill constraint-based mutants than the AVM.

3) Mutation Score: Figure 6 gives the number of database interactions ($|D|$) and the mutation score (M_D) resulting from the application of *SchemaAnalyst* and *DBMonster* to the 25 schemas. The upper bar chart shows that AVM may achieve higher mutation scores when the schemas are hosted by HSQLDB and SQLite, highlighting the fact that the DBMSs interpret schema-creating statements differently and suggesting that a multi-DBMS approach to evaluation is important in practice. Since *DBMonster* only worked correctly with Postgres, the remainder of the analysis focuses on both method's scores for this DBMS.

For a simple schema without constraints, like Cloc, the AVM generates four database interactions that achieve a mutation score slightly above 0.6. Across all schemas, *DBMonster* achieves its highest mutation score for Cloc by always generating 50 or more database interactions that yield a median value that is also near 0.6. This result reveals that, while *DBMonster* is an acceptable method for testing relational schemas that contain few, if any, constraints, it still does so in a way that is less cost-effective than *SchemaAnalyst*. In contrast to *SchemaAnalyst*, *DBMonster* is also unable to generate effective data to test complex, real-world schemas like JWhoisServer: for this schema, the AVM earns a mutation score greater than 0.7 by generating 62 `INSERT`s while the random search produces over 300 database interactions that lead to a score near 0.2.

It is also interesting to observe that *DBMonster* generates a median number of 13,647.5 database interactions that lead to a mutation score between 0.4 and 0.6 for the NistDML181 schema. In contrast, *SchemaAnalyst* achieves a comparable score of 0.6 with only seven database interactions – a result that is more cost-effective and in better support of testing and debugging. Finally, while *DBMonster* fails to generate schema-compliant data for the CustomerOrder, Flights, NistDML182, NistXTS748, Person, and RiskIt schemas, the AVM produces effective data for every schema. Across all of the schemas, *SchemaAnalyst*'s descriptive statistics of (0.29, 0.59, 0.65, 0.70, 0.89) suggest that, while sometimes it kills less than 30% of the mutants, it kills a median of more than 65% and, for certain schemas like DellStore, it can kill nearly 90% of the mutants. Factoring in a kill score of zero for the six schemas for which it crashes, *DBMonster*'s statistics of (0.0, 0.11, 0.41, 0.52, 0.68) reveal that, according to the M_D metric, it is less effective than the AVM.



NistDML181 not visible in the left graph due to the magnitude of its database interaction count. (Min: 9730.0, Median: 13647.5, Max: 17390.0)
DBMonster fails to generate data for CustomerOrder, Flights, NistDML182, NistXTS748, Person, and RiskIt

The upper graph is a bar chart because *SchemaAnalyst*'s mutation scores did not vary across the 30 trials. The lower graph is a box-and-whisker plot because *DBMonster*'s values for $|D|$ and M_D differ considerably over the trials.

Figure 6. Number of Database Interactions and Mutation Score for AVM (Upper) and DBMonster (Lower) for All of the Schemas.

4) Efficiency: Even though HSQLDB and SQLite are in-memory databases, they still access the disk during data generation and seem to do so in a way that is much less efficient than Postgres. Yet, we note that generation with Postgres being faster than with the other DBMSs is less important than the fact that data generation performance varies across multiple DBMSs. Since data generation efficiency is only a concern if a technique incurs a high time overhead, we briefly highlight the trade-offs between the methods.

With the exception of Products, for which *DBMonster* is faster by about five seconds, the AVM and *DBMonster* require a similar amount of time to generate data for the simple schemas (e.g., AVM takes one to two seconds and *DBMonster* uses less than five to produce data for Inventory). For more complex schemas that contain more tables and constraints, both methods incur a higher overhead. Yet, while the AVM needs no more than 25 seconds and 3 seconds to respectively generate data for CustomerOrder and JWhoisServer, *DBMonster* respectively takes up to 30 seconds and 10 seconds for the same schemas.

When it comes to efficiency, it is also important to note that *DBMonster* exhibits more variability than *SchemaAnalyst* – it takes between 15 and 30 seconds to generate data for Flights while AVM never needs more than 2 seconds for this schema. Unlike *SchemaAnalyst*, *DBMonster* takes a median time of 634 seconds while attempting to generate data for the NistDML182 schema that contains a compound foreign key composed of 15 columns. In contrast, AVM creates data that satisfies and negates NistDML182's constraints in less than two seconds. With descriptive statistics of (0.41, 1.09, 1.90, 5.07, 36.52) and (1.50, 3.01, 5.21, 16.79, 639.93) for *SchemaAnalyst* and *DBMonster*, respectively, it is clear that the AVM exhibits time overheads that are competitive with the random method.

E. Threats to Validity

Although the results show that *SchemaAnalyst* is as efficient as *DBMonster* and more capable of automatically generating small data sets that cover constraints and kill mutants in a cost-effective manner, there are threats to the validity of this paper's empirical evaluation. In order to mitigate the threat associated with the generalization of the results, we picked 25 representative schemas that exhibited different data types and integrity constraints, as previously discussed in Section V-A. The fact that a stochastic search algorithm may produce a different result for each of its runs means that this paper's empirical results may not be representative of the chosen methods' general behavior. Since both the AVM and *DBMonster* perform random actions, we ran them for 30 trials, as mentioned in Section V-B.

If *DBMonster* is not representative of the database generators commonly used in practice, then this could also be a threat to validity. Even though *DBMonster* employs a random search and is not specifically designed for testing and debugging purposes, it is comparable to *SchemaAnalyst* since both tools are supposed to handle the most common types of integrity constraints. Real-world database designers and testers use *DBMonster* because, unlike potentially expensive commercial tools [18], it is free and open source and documented by several online tutorials. Moreover, since "check constraints are hardly ever considered" by commercial database generation tools [18], *DBMonster* is not likely to be worse than these more costly options. Even though the results in Section V-D show that *SchemaAnalyst* is superior to *DBMonster*, we judge that the comparison of these tools is appropriate. Finally, we controlled threats arising from defects in the tools themselves by carefully testing *SchemaAnalyst* and manually checking the results on a wide variety of simple schemas.

VI. Related Work

Due to space constraints and the fact that *SchemaAnalyst* is unique in its focus on data generation for testing relational schema integrity constraints, we briefly survey the related work. Like *SchemaAnalyst*, many methods (e.g., [1], [2], [3], [4], [9], [15]) attempt to generate data, with Chays et al. and Houkjær et al. presenting partially-automated methods [4], [15] and the other papers describing automatic data generators [1], [2], [3], [9]. Our schema testing technique is complementary to these approaches. Like this paper, other work has considered database-aware mutation analysis (e.g., [8], [19]). Yet, Tuya et al. exclusively focus on the SQL `SELECT` statement [8] and Chan et al. neither describe an implementation nor furnish an empirical evaluation [19].

VII. Conclusions and Future Work

The correctness of a relational database's schema and the integrity constraints that it expresses are a critical aspect of overall database application correctness. This paper presents and empirically evaluates *SchemaAnalyst*, a search-based method for efficiently and effectively testing relational integrity constraints. In support of the testing of real-world schemas, *SchemaAnalyst* can model and handle the data types and constraints associated with three representative DBMSs: Postgres, HSQLDB, and SQLite. Beyond presenting an AVM for generating a test suite that systematically satisfies and negate the schema's constraints, this paper describes an approach to schema mutation.

An empirical study with 25 database schemas reveals that *SchemaAnalyst* can efficiently generate data where *DBMonster* either crashes or incurs an unacceptably high time overhead. Since *SchemaAnalyst* attains higher constraint coverage and mutation scores than *DBMonster*, we will, as part of future work, enhance it by employing both search-based methods and constraint solvers and by applying it to extra schemas using additional configurations.

Acknowledgements: Phil McMinn is funded in part by EPSRC grant EP/I010386/1.

References

[1] C. Binnig, D. Kossmann, and E. Lo, "Multi-RQP: Generating test databases for the functional testing of OLTP applications," in *Proc. of TDS*, 2008.

[2] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya, "Constraint-based test database generation for SQL queries," in *Proc. of AST*, 2010.

[3] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Proc. of ASE*, 2011.

[4] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA for testing relational database applications," *Soft. Test., Verif. and Reliab.*, vol. 14, no. 1, 2004.

[5] S. Abdul Khalek and S. Khurshid, "Automated SQL query generation for systematic testing of database engines," in *Proc. of ASE*, 2010.

[6] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proc. of VLDB*, 2007.

[7] P. McMinn, "Search-based software test data generation: A survey," *Soft. Test., Verif. and Reliab.*, vol. 14, no. 2, 2004.

[8] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Mutating database queries," *Infor. and Soft. Tech.*, vol. 49, no. 4, 2006.

[9] S. Khalek, B. Elkarablieh, Y. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *Proc. of ASE*, 2008.

[10] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "QAGen: generating query-aware test databases," in *Proc. of SIGMOD*, 2007.

[11] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *Proc of ICST*, 2010.

[12] B. Korel, "Automated software test data generation," *IEEE Trans. on Soft. Eng.*, vol. 16, no. 8, 1990.

[13] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global and hybrid search," *IEEE Trans. Soft. Eng.*, vol. 36, 2010.

[14] L. Bottaci, "Type sensitive application of mutation operators for dynamically typed programs," in *Proc. of Mutation*, 2010.

[15] K. Houkjær, K. Torp, and R. Wind, "Simple and realistic data generation," in *Proc. of VLDB*, 2006.

[16] J. Cobb, J. A. Jones, G. M. Kapfhammer, and M. J. Harrold, "Dynamic invariant detection for relational databases," in *Proc. of WODA*, 2011.

[17] D. C. Hoaglin, F. Mosteller, and J. W. Tukey, *Understanding robust and exploratory data analysis*. Wiley, 2000.

[18] K. Haller, "The test data challenge for database-driven applications," in *Proc. of TDS*, 2010.

[19] W.K. Chan, S.C. Cheung, and T.H. Tse, "Fault-based testing of database application programs with conceptual data model," in *Proc. of QSIC*, 2005.