This is a repository copy of *SchemaAnalyst: Search-Based Test Data Generation for Relational Database Schemas*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/104205/

Version: Accepted Version

**Proceedings Paper:**

https://doi.org/10.1109/ICSME.2016.93

© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

**Reuse**
Unless indicated otherwise, fulltext items are protected by copyright with all rights reserved. The copyright exception in section 29 of the Copyright, Designs and Patents Act 1988 allows the making of a single copy solely for the purpose of non-commercial research or private study within the limits of fair dealing. The publisher or other rights-holder may allow further reproduction and re-use of this version - refer to the White Rose Research Online record for this item. Where records identify the publisher as the copyright holder, users can verify any specific terms of use on the publisher's website.

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.
SchemaAnalyst: Search-Based Test Data Generation for Relational Database Schemas

Phil McMinn and Chris J. Wright
University of Sheffield

Cody Kinneer, Colton J. McCurdy, Michael Camara, and Gregory M. Kapfhammer
Allegheny College

Abstract—Data stored in relational databases plays a vital role in many aspects of society. When this data is incorrect, the services that depend on it may be compromised. The database schema is the artefact responsible for maintaining the integrity of stored data. Because of its critical function, the proper testing of the database schema is a task of great importance. Employing a search-based approach to generate high-quality test data for database schemas, SchemaAnalyst is a tool that supports testing this key software component. This presented tool is extensible and includes both an evaluation framework for assessing the quality of the generated tests and full-featured documentation. In addition to describing the design and implementation of SchemaAnalyst and overviewing its efficiency and effectiveness, this paper coincides with the tool’s public release, thereby enhancing practitioners’ ability to test relational database schemas.

I. INTRODUCTION

Healthcare, science, and commerce often rely on information that is stored in databases [1]. When this data is incorrect, passengers can have their flights delayed or patients may receive the wrong medication [2]. In addition to documenting the structure of and connections between data entries, relational databases furnish means for protecting the correctness of the data that they store. In particular, the relational database schema is the artefact that is responsible for safeguarding the integrity of a database. The crucial role of the database schema makes the testing of it a task of vital importance.

While non-relational “NoSQL” database systems have been gaining in popularity, relational databases remain pervasive. For instance, Skype, the widely used video-call software, uses the PostgreSQL database management system (DBMS) [3] while Google makes use of the SQLite DBMS in Android-based phones [4]. Moreover, according to DB-Engines.com, the three most popular DBMSs are relational in nature [5]; also, the 968,277 questions asked on StackExchange about relational databases show the demand for their support [6].

SchemaAnalyst is a tool for generating high-quality test data in support of database schema testing. Using a search-based approach that incrementally improves a test suite by repeated fitness evaluations, SchemaAnalyst discovers data instances that comprehensively exercise a database schema hosted by either HyperSQL, PostgreSQL, or SQLite [7]. It also includes an evaluation framework with a collection of real-world schemas, as well as a mutation analysis system that enables verifying the quality of the generated test data based on its capability to detect systematically seeded faults. Also, SchemaAnalyst is extensible, well documented, and available for download [8].

SchemaAnalyst has been used to support research studies focusing on both search-based software testing [7], [9], [10] and mutation testing [11], [12], [13], [14]. In addition to describing the design and implementation of SchemaAnalyst and overviewing its efficiency and effectiveness, this paper inaugurates the public release of this tool. Since past studies have shown the benefits of using the presented open-source tool instead of competing systems [7], this paper argues that SchemaAnalyst is ready to enhance practitioners’ testing of schemas. In summary, the key contributions of this paper are:

1) SchemaAnalyst, an extensible, efficient, and effective tool that generates test data for database schemas (Section III).
2) In support of researchers, a comprehensive evaluation framework, including relational schemas suitable for further empirical study and mutation analysis tools supporting the assessment of test data quality (Section III).
3) Aiding both researchers and practitioners, documentation explaining the features and usage of the tool (Section IV).
4) Confirming SchemaAnalyst’s scalability and applicability, a survey of relevant empirical results (Section V).

II. BACKGROUND

Software testing, the process of running a software system to ensure that it functions as intended, is a key part of the software development lifecycle [15]. Software that produces unexpected output contains a fault. Developers can check for these faults by running test cases that give the program inputs and check for expected outputs [16]. If the software produces the expected output for the provided input, then this suggests that it is functioning correctly. Yet, if it does not perform as anticipated, then the tests may have found a fault.

A collection of test cases is called a test suite. A test suite’s effectiveness at finding faults is known as its adequacy, which is assessed by a test suite adequacy criterion. Writing high-quality tests requires developers to painstakingly consider the range of possible inputs — as anticipated, this is a challenging and time-consuming process [17]. Test data generation reduces the burden on a human tester by (semi-)automatically creating test inputs. As described in this paper, search-based test data generation with SchemaAnalyst employs a fitness function to direct the tool towards creating high-quality test data [18].

Mutation adequacy is a criterion that measures the effectiveness of a test suite by modifying the artefact under test to produce a “mutant” [19]. This change to the entity is meant to simulate a potential fault, so that the mutant should result in behavior different from that of the original. In this process, the result from running the tests against the original and mutant artefacts are compared. If the results are the same, then the test suite failed to detect the seeded fault. Yet, if they are different,
CREATE TABLE Inventory  
   (id INT PRIMARY KEY,  
   product VARCHAR(50) UNIQUE,  
   quantity INT,  
   price DECIMAL(18,2)  
);  

Fig. 1. The Inventory relational database schema

then the test suite found the simulated fault, at which point the 
migrant is said to be “killed”. The mutation score is the number of 
mutants killed divided by the total number of mutants [20].

Managed by applications called database management sys-
tems (DBMSs), a relational database is a collection of con-
ected data [2]. The database schema is the artefact that lays 
out the structure of the database, organising it into tables and 
columns. The schema can also define integrity constraints, or 
rules that the candidate data must meet before the DBMS will 
accept it. If the pending data violates an integrity constraint 
specified by the schema, then the DBMS rejects it as invalid.

Figure 1 furnishes a database schema for recording the number 
of products kept in an inventory. This schema defines one 
table, called Inventory, with four columns. The id column 
on line 3 is annotated with the PRIMARY KEY constraint, 
indicating that data inserted into it cannot be left missing or 
unknown, and that the values in this column must be unique. 
If the PRIMARY KEY was left out of the database schema, 
then multiple items could be entered with the same id value, 
potentially resulting in incorrect application behavior.

III. THE SchemaAnalyst TEST DATA GENERATION TOOL

An error in the specification of the database schema may 
result in the corruption of the data state and the disruption 
of a supported service. Even though verifying the accuracy of 
the database schema is a critical step towards protecting data 
integrity, using manually created test data to do so is often time 
consuming and error prone [7]. Figure 2 provides a high-level 
overview of this paper’s tool that uses a search-based approach 
to automatically generate tests for database schemas.

After being given a schema as input, SchemaAnalyst uses a 
coverage criterion to systematically create a collection of test 
requirements, or the rules that the test data must try to fulfill. 
One example of a coverage criterion is Integrity Constraint 
Coverage (ICC) [9], which has two test requirements for 
every integrity constraint in the schema: one requiring that the 
constraint is satisfied and another necessitating its violation. 
So, a test requirement for the Inventory schema in Figure 1 
might be “the PRIMARY KEY constraint on line three must be 
violated”. Running a test that fulfilled this requirement would 
draw attention to the fact that, for instance, SQLite allows 
a PRIMARY KEY to be NULL, unlike most other DBMSs. 
SchemaAnalyst supports 9 coverage criteria in total [9].

Using a test data generator, the presented tool creates 
test data to satisfy the test requirements; the default test 
data generator used by SchemaAnalyst is based on Korel’s 
Alternating Variable Method (AVM) [21]. This data generator 
uses a fitness function to evaluate how well the test data 
satisfies the requirements, thus aiding it in producing test 
data that satisfies more of the requirements [9]. For example, 
SchemaAnalyst generated the following INSERT statement to 
violate the PRIMARY KEY constraint on line three in Figure 1. 

```
INSERT INTO Inventory VALUES (NULL, '', 0, 0);
```

The version of SchemaAnalyst presented in this paper en-
codes this test case in the well-established JUnit format that 
is commonly used by developers, thereby providing a way to 
easily apply SchemaAnalyst to industrial databases. The JUnit 
test first runs the INSERT statement on an installed DBMS 
(e.g., SQLite) and then asserts that the INSERT statement was 
rejected by the schema. If this is the case, then the test passes. 
If the schema allows the bad data, then the test case fails.

Although not included in Figure 2 due to space constraints, 
SchemaAnalyst also includes features to evaluate the quality 
of the generated test data. First, it calculates coverage, or 
the percentage of the requirements satisfied by the test data. 
Test data quality can also be measured using the provided 
mutation testing tools. When executed in mutation testing 
mode, SchemaAnalyst will generate mutant database schemas 
and compare the behavior of the test suite on the original and 
mutant schemas. SchemaAnalyst includes 14 different mutation 
operators that can be used to assess test suite quality [13].

IV. SchemaAnalyst’S DESIGN AND IMPLEMENTATION

A. Design

SchemaAnalyst is implemented in the Java programming 
language. Designed with extensibility in mind, the tool is 
divided into 13 packages, which this paper briefly overviews. 
The sqlrepresentation package provides an interme-
diate Java representation of data structures in relational 
databases, fully modelling database tables, columns, expres-
sions, data types, integrity constraints, and other relevant 
entities. These objects enable SchemaAnalyst to support mul-
tiple DBMSs (i.e., SQLite, PostgreSQL, and HyperSQL), and, 
aditionally, allow for the inclusion of new DBMSs. The tool
also contains the sqlparser package that wraps the General SQL Parser [22], thus enabling the effective conversion of a schema expressed in the Structured Query Language (SQL) to the tool’s intermediate representation. As this SQL parser is a commercial product, the open-source version of SchemaAnalyst does not provide it for download. Therefore, users can experiment with SchemaAnalyst by either testing the provided schemas or (automatically or manually) converting a new schema to the tool’s internal SQL representation.

The test generation package provides a representation of test suites and test cases, along with test requirements and the 9 coverage criteria [9]. The data package furnishes the 3 test data generators, as well as various generic data-type representations for use during test data generation [9]: Table I summarizes the coverage criteria and data generators furnished by the version of SchemaAnalyst presented in this paper.

The dbms package provides support for three DBMSs and includes the classes that enable interaction with an installed DBMS. The sqlwriter package furnishes support for creating SQL statements for use with DBMSs and is used with the javawriter package to encode the generated test data as a JUnit test suite. The mutation package provides the mutation analysis functionality, including the mutation operators [13], mutant equivalence and reduction features [12], and means for performing mutation analysis [14].

B. Usage Instructions

SchemaAnalyst is publicly available on GitHub under an open-source license [8]. After cloning the Git repository, the project can be built using Gradle by running the following command in the project’s root directory: ./gradlew compile. After the tool compiles, the user must set the CLASSPATH so that it contains build/classes/main, build/lib/*, lib/* and the current working directory.

Optionally, the user can install the PostgreSQL, SQLite, and HyperSQL DBMSs. Since SQLite does not require configuration on the computer running SchemaAnalyst, it is currently the default option. Using the chosen DBMS, SchemaAnalyst will run the generated test suite. If the use of an actual DBMS is desired, the user should refer to online documentation for detailed instructions [8]. The tool also supports a “virtual” DBMS executor allowing SQL statements to be simulated.

Usage instructions for SchemaAnalyst can be obtained by running java org.schemaanalyst.util.Go --help; Figure 3 shows a snippet of this menu. As indicated by the help display, SchemaAnalyst first expects options indicating the desired schema, coverage criterion, data generator, and DBMS. Defaults are provided for all of these options except for the schema option, which is required. The user must then give a command. The two supported commands are generation, used to generate test data, and mutation, used to evaluate the quality of test data. To run SchemaAnalyst to generate test data for the provided inventory schema, the following command could be used: java org.schemaanalyst.util.Go -s parsedcasesstudY.inventOry generation.

With no other command-line options, SchemaAnalyst will produce a Java class containing a JUnit test suite with the generated test data. By default, this class will be created under the generatedtest package and saved in a directory of the same name in the tool’s root directory. The user may append the --inserts option to the generation command to obtain the generated test data in the form of SQL INSERT statements that are saved in plain text instead of a JUnit test suite.

If the mutation command is given to perform mutation analysis on the test data generated by SchemaAnalyst, a directory called results will be created in the project’s root directory. It will contain a comma-separated value file recording the parameters used in the analysis as well as the mutation score and some additional runtime information.

The SchemaAnalyst GitHub page also provides comprehensive documentation, including installation and usage instructions that detail the inputs, outputs, and behavior of the tool [8]. In addition to featuring a thorough JUnit test suite, the source code of SchemaAnalyst contains documentation to aid developers who want to extend the tool or to better understand certain implementation decisions. As an overall contribution of this paper, the presented tool’s GitHub repository now includes over 52,000 lines of Java code and tens of thousands of lines of scripts and SQL code that enable others to try the provided examples of schema testing, reproduce the results from our prior experiments, and apply SchemaAnalyst to new schemas.

V. APPLYING THE SCHEMAANALYST TOOL

As shown in Table II, SchemaAnalyst has been used in several prior experimental studies, thereby facilitating research into both search-based testing and mutation testing. To date, published papers about the presented tool report on using it to
test 35 relational schemas, including those from databases in
off-used open-source software. Houkjær et al. [23] note that
real-world complex relational schemas often include features
such as composite keys and multi-column foreign-key rela-
tionships. As such, the schemas chosen for past studies reflect
a diverse set of features from simple instances of every main
type of integrity constraint (i.e., PRIMARY KEY constraints,
FOREIGN KEY constraints, UNIQUE constraints, NOT NULL
constraints, and CHECK constraints) to more complex exam-
ples involving many-column foreign key relationships.

Several schemas used in past studies were taken from
real-world database-centric applications: JWhoisServer is used in
an open-source, Java-based implementation of a server
for the Internet “WHOIS” protocol (http://jwhoisserver.net). Both MozillaExtensions and MozillaPermissions were ex-
tracted from SQLite databases that are a part of the Mozilla
Firefox Internet browser. RiskIt is part of system for modeling
the risk of insuring individuals (http://sourceforge.net/projects/
riskitinsurance), while StackOverflow is the schema used by
a popular programming question and answer website, as
previously studied in a conference data mining challenge [24].
Some of these schemas have been featured in previous studies of
various testing methods (e.g., RiskIt and UnixUsage [25], and
JWhoisServer [26]). ArtistSimilarity and ArtistTerm are part of
the “Million Song” dataset, a database of song metadata [27].
It is worth noting that SchemaAnalyst’s GitHub repository
currently furnishes 95 schemas, including those that are deriva-
tives of the main schemas and thus ideal for testing purposes.

Due to space constraints, the remainder of this section
overviews experimental studies of SchemaAnalyst’s capabil-
y to automatically generate test data; readers interested in other
related work can read the 6 papers referenced in Table II.
Kapflhammer et al. compared SchemaAnalyst to DBMonster in
an experiment using mutation testing for 3 DBMSs and 25
database schemas [7]. The results showed that SchemaAna-
lyst outperformed DBMonster in terms of mutation score and
constraint coverage, while remaining competitive in execution
time. McMinn et al. organized the 9 coverage criteria used in
SchemaAnalyst into a subsumption hierarchy and, addition-
ally, investigated the effectiveness of the criteria in a study
using 3 DBMSs and 2 database schemas [9]. The results
showed mutation scores as low as 12% for the least stringent
criteria and as high as 96% for the most stringent. Kinneer et
al. studied the scalability of SchemaAnalyst, finding that the
tool scaled well for all realistically sized schemas [10], [28].
Kinneer also enhanced SchemaAnalyst to generate test data
for both relational database queries and schemas, providing
evidence of SchemaAnalyst’s extensibility [29]. Finally, Mc-
Curdy et al. used the results from SchemaAnalyst’s mutation
analysis of schemas to support selective mutation testing [30],
showing that the presented tool can integrate with other tools.

VI. CONCLUSIONS AND FUTURE WORK

Many database-centric services rely on the quality of the
underlying data. Much of this data is managed by relational
databases, with the database schema protecting the integrity
of the data. Testing the schema for correctness is vital to ensuring
data quality. SchemaAnalyst is a tool that generates test data
for a relational database schema, thereby increasing confidence
in the schema’s correctness. Using a search-based technique,
SchemaAnalyst automatically creates high-quality test data
across multiple DBMSs. The presented tool also includes an
evaluation framework that provides 95 case-study schemas and
support for efficient mutation analysis. In addition to being
used in 6 published studies, the presented tool is now available
from http://www.schemaanalyst.org [8]. With an open-source
license and a modular design, SchemaAnalyst is an extensible
tool for search-based test data generation and mutation testing,
enabling the work of both researchers and practitioners.

In future work, we will evaluate how SchemaAnalyst helps
the people who design and test database schemas. We plan
to incorporate techniques that generate more readable and
realistic data values [31], [32], [33], [34], thus helping humans
understand test cases more easily [17], [35]. We will also in-
tegrate the tool with others that support software maintenance
activities like regression testing [36] and fault localization [37].
Next, we will extend the tool so that it enables the testing of
recently developed NoSQL systems. Ultimately, the current
version of SchemaAnalyst, our planned extensions, and the
features and studies contributed by the new researchers and
industrialists using this now-released tool will yield a com-
prehensive approach to testing database-centric applications.
REFERENCES


[34] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does automated white-box test generation really help software testers?” in Proc. of ISTA.
