Automated Software Testing of Relational Database Schemas

Abdullah Alsharif

Supervisor: Professor Phil McMinn
External Advisor: Dr Gregory Kapfhammer

A thesis submitted in partial fulfilment of
the requirements for the degree of Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

February 2020

I would like to dedicate this thesis to my beloved parents, grandparents (May Allah grant them paradise), and foremost my loving wife.

Abstract

Relational databases are critical for many software systems, holding the most valuable data for organisations. Data engineers build relational databases using schemas to specify the structure of the data within a database and defining integrity constraints. These constraints protect the data's consistency and coherency, leading industry experts to recommend testing them.

Since manual schema testing is labour-intensive and error-prone, automated techniques enable the generation of test data. Although these generators are well-established and effective, they use default values and often produce many, long, and similar tests — this results in decreasing fault detection and increasing regression testing time and testers inspection efforts. It raises the following questions: How effective is the optimised random generator at generating tests and its fault detection compared to prior methods? What factors make tests understandable for testers? How to reduce tests while maintaining effectiveness? How effectively do testers inspect differently reduced tests?

To answer these questions, the first contribution of this thesis is to evaluate a new optimised random generator against well-established methods empirically. Secondly, identifying understandability factors of schema tests using a human study. Thirdly, evaluating a novel approach that reduces and merge tests against traditional reduction methods. Finally, studying testers' inspection efforts with differently reduced tests using a human study.

The results show that the optimised random method efficiently generates effective tests compared to well-established methods. Testers reported that many `NULL`s and negative numbers are confusing, and they prefer simple repetition of unimportant values and readable strings. The reduction technique with merging is the most effective at minimising tests and producing efficient tests while maintaining effectiveness compared to traditional methods. The merged tests showed an increase in inspection efficiency with a slight accuracy decrease compared to only reduced tests. Therefore, these techniques and investigations can help practitioners adopt these generators in practice.

Acknowledgement

First and foremost, all praises to Allah the Merciful for the strengths and blessing in completing this research.

I want to express my deep and sincere gratitude to my supervisor, Professor Phil McMinn, who has been a constant source of knowledge, advice, inspiration, and support to achieve the accomplishments within this thesis. Special thanks to my external advisor, Dr Gregory Kapfhammer, for his support, encouragement, patience, and his hospitality while being in the USA.

I would also like to thank the following friends and colleagues for their great support and in no particular order: Ibrahim Althomali, Nasser Albanian, Mathew Hall, David Paterson, Tom White, José Miguel Rojas, Abdulaziz Almanea, Anas Alsuhaibani, Michael Foster, Ben Clegg, Donya Yazdani, Sina Shamshiri, Alasdair Warwicker, Edgar Covantes Osuna, Jorge Pérez Heredia, George Hall, José Campos, Adam and Alison Poulston. I am grateful to the members of both the Verification and Testing groups for their encouragement and fun lunches. I would also like to thank my friends who I met in Sheffield and made my journey a great experience.

I will be eternally grateful to my parents for their unconditional love and for encouraging me all the way. My gratitude also goes to my family members for their support and encouragement during my studies. I am also grateful for my wife, Shahad, who supported throughout my studies.

Publications

Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. DOMINO*: Fast and Effective Test Data Generation for Relational Database Schemas.* In Proceedings of the 11th International Conference on Software Testing, Validation and Verification (ICST 2018), 2018

Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. *Running Experiments and Performing Data Analysis using SchemaAnalyst and* DOMINO. Presented at the 11th International Conference on Software Testing, Validation and Verification (ICST 2018) – Tool Demo Track

Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. *Generating Test Suites With DOMINO.* Presented at the 11th International Conference on Software Testing, Validation and Verification (ICST 2018) – Tool Demo Track

Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. *What Factors Make SQL Test Cases Understandable For Testers? A Human Study of Automatic Test Data Generation Techniques.* In Proceedings of the 35th International Conference on Software Maintenance and Evolution (ICSME 2019)

Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. *STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases.* In Proceedings of the 13th International Conference on Software Testing, Verification and Validation (ICST 2020)

Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. *Hybrid Methods for Reducing Database Schema Test Suites: Experimental Insights from Computational and Human Studies.* In Proceedings of the 1st International Conference on Automation of Software Test (AST 2020)

Contents

Glossary of Terms

AVM	Alternating Variable Method is a search-based technique. 16
SQL	Structured Query Language. 25
ClauseAICC	Clause-Based Active Integrity Constraint Coverage criterion. 39
AUCC	Active Unique Column Coverage criterion. 41
ANCC	Active Null Column Coverage criterion. 41
AVM-D	Alternating Variable Method that starts with Default Values. 46
Random$^+$	A random test data generator that utilises a pool of constants. 47
HGS	Harrold, Gupta, and Soffa Reduction Technique. 59
AVM-R	Alternating Variable Method that starts with Random Values. 67
DOMINO	DOMain-specific approach to INtegrity cOnstraint test data generation. 68
DOM-RND	DOMINO that generate random values. 68
AVM-LM	Alternating Variable Method that utilises Language Model values. 93
DOM-COL	DOMINO that utilises column name as string values for test data. 94
DOM-READ	DOMINO that utilises dictionary values to generate test data. 94
STICCER	Schema Test Integrity Constraints Combination for Efficient Reduction. 121
STICCER-HGS	A STICCER variant that merge test suites reduced with HGS. 141

Chapter 1

Introduction

1.1 Overview

Data is highly important in many areas (e.g., banks, health, taxes, and many more) that enables people and businesses to make informed decisions. Such influence can affect people's lives. For example, medical test results (i.e., data) that impacts the doctor's judgment could affect a patient's health [1]. Hence, data must be available and correct for such crucial situations.

Computer systems store and manipulate data using Database Management Systems (DBMSs) through a database instance. DBMSs allow data engineers to design and build databases that are either categorised relational (e.g., SQL) or non-relational (e.g., NoSQL). The most popular type are relational databases [2] and they are engineered using "blueprints" which called *schemas*. Thus, each relational database instance must include a defined schema that define the data structure that enforce how the data is stored within a database. A schema must described in advance with defined set of rules and relations. Such as the data structure, relationship between data, and the permissible data types [3]. This ensures the DBMS stores and retrieves data while abiding the user's defined schema. However, wrongly designed schema can allow unintended storing of incorrect or corrupt data within the database [4]. Therefore, it might lead users to make wrong assumptions that influence crucial decisions.

Relational database management systems (RDBMSs), such as SQLite [5] or PostgreSQL [6], are enterprise database engines that administer one or more of relational databases, each specified by a schema. Hence, a schema can be constructed using the Structured Query Language (SQL) with "`CREATE TABLE`" statements, such as those shown in Figure 1.1. This `CREATE TABLE` statement describes a table called `person`, for storing rows of data where each row corresponds to a person. Nested within the `CREATE TABLE` statement is a list of *columns* ("`id`" to "`date_of_birth`") that describe the data to be stored in each row about each person, including first and last names, and gender. Each column has an associated type, such as a string (represented using the SQL "`VARCHAR`" type), integer, and date.

```
CREATE TABLE person (
  id int NOT NULL,
  last_name varchar(45) NOT NULL,
  first_name varchar(45) NOT NULL,
  email varchar(45) NOT NULL UNIQUE,
  gender varchar(6) NOT NULL,
  date_of_birth date NOT NULL,
  PRIMARY KEY(id)
  CHECK (gender IN ('Male', 'Female', 'Other'))
);
```

Figure 1.1: The illustration of a relational database schema

Underneath the definition of columns (and occasionally in-line with a column definition — for example, the `UNIQUE` on the `email` column) are the declarations of *integrity constraints* on the data in the table. For instance, the `PRIMARY KEY` integrity constraint enforces the `id` column value to be unique for each row of the table. Similarly, the `email` must be unique as guarded by a `UNIQUE` constraint. Columns marked `NOT NULL` mean that undefined values using `NULL` are not allowed to be stored for such columns. Finally, the defined `CHECK` constraint only allows one value from the defined list to be selected (i.e., "Male", "Female", or "Other") and stored within the `gender` column.

These constraints are significant because they protect the consistency and coherency of data in a relational database. While a correct schema design ensures the integrity of the data within the database, inadvertent definitions (i.e., omitting constraints or adding the wrong constraints) can manifest in a failure that corrupts the data [7]. Such as not having a `UNIQUE` constraint on the `email` column will allow the multiple persons to be stored with one email address. Forgetting to define a `NOT NULL` on a column might lead to issues, such as a bank account without a person's name or not having a starting date for an employee [8]. A developer can also unintentionally add an integrity constraint, such as a `UNIQUE` constraint on a column representing somebody's first name. Such mistakes may happen in combination, as the unique constraint may have been intended for a column representing some distinctly identifiable information, such as an identification number.

1.2 Motivation

Even though many tutorials explain how to avoid making mistakes when designing a relational database schema (e.g., [9, 10, 11, 12]), data engineers may incorrectly specify or omit integrity constraints. Since RDBMSs often interpret the SQL standard differently, a schema may exhibit a different behaviour during development and after deployment. Therefore, as advocated by industrial practitioners [13], it is necessary to test the schema's integrity constraints to ensure that behaves as to what an engineer expects.

Since haphazard methods may overlook schema faults, McMinn et al.'s work [7] presented a family of coverage criteria that support systematic testing. These criteria require the creation of specific rows of database data that, when featured in SQL `INSERT` statements, exercise integrity constraints as *true* or *false*, or, test some particular property of the constraint. Frequently, to satisfy these coverage requirements, certain values may need to be identical to one another, different, or `NULL`, across different rows of data. For example, to violate a primary key (i.e., to exercise it as *false*), two rows of data need to be created with the same values for the key's columns. To satisfy a `UNIQUE` constraint (i.e., exercise it as *true*), values across rows need to be different. To violate a `NOT NULL` constraint, a particular column must be `NULL`. Manual tests can be created as in Figure 1.2 that shows a tester trying to exercise the `UNIQUE` constraint of the `person` schema (in Figure 1.1) as true (`INSERT`s that satisfy the integrity constraints in part (a)) and as false (`INSERT`s that violate the `UNIQUE` constraint in part (b)).

Since it is challenging for a tester to cover all of these requirements man-

```
1) CREATE TABLE person ( .....  );                                          ✓
2) INSERT INTO places(id, last_name, first_name, email, gender)
   VALUES (1, 'John', 'Doe', 'john@example.com', 'Male');                    ✓
3) INSERT INTO places(id, last_name, first_name, email, gender)
   VALUES (2, 'Jane', 'Doe', 'jane@example.com', 'Female');                  ✓
4) DROP TABLE person;                                                        ✓
```

(a) Test Case 1 exercise the `person` table as true

```
1) CREATE TABLE person ( .....  );                                          ✓
2) INSERT INTO places(id, last_name, first_name, email, gender)
   VALUES (1, 'John', 'Doe', 'john@example.com', 'Male');                    ✓
3) INSERT INTO places(id, last_name, first_name, email, gender)
   VALUES (2, 'Jane', 'Doe', 'john@example.com', 'Female');                  ✗
4) DROP TABLE person;                                                        ✓
```

(b) Test Case 2 exercise `person` table as false

Figure 1.2: Example of test cases for the `person` table in Figure 1.1. Each tests must be ran on an empty database. The ✓ and ✗ illustrate that the database will accept or reject the inserted data, respectively.

ually (i.e., exercising each integrity constraint), McMinn et al. [14] created a tool that automatically generates the tests. The tool generates test data to satisfy each test requirement. The test requirements are also automatically generated with a given schema and coverage criteria. The state-of-the-art test data generator is based on the Alternating Variable Method (AVM) [15], a search-based method that receives guidance from a fitness function [16, 7]. However, the generation of test data with a search technique and random values can be slow because it will be searching for identical values (e.g., matching `PRIMARY KEY` values) and slowly adjusting those values until they are the same. To aid the process, the AVM may be configured to start with a series of "default" values that ensure matches are likely from the outset. However, this can introduce many similarities across different tests in the suite, hindering both its diversity and potential fault-finding capability [7, 17].

It is also challenging to create test cases that are understandable and maintainable [18, 19] — mainly when the tests use complex and inter-dependent `INSERT` statements to populate a relational database [20]. While automated test data generators can create test cases that aid systematic database schema testing [16], the human cost associated with inspecting test output and understanding test outcomes is often overlooked [21]. Also, tests should be understandable for inspection, primarily when database schemas evolve [22] and new tests are automatically generated. This understandability is subjective, with developers having different views of automatically generated tests [23]. For example, if testers are deciding whether or not the database will reject a test, some may prefer English-like strings, while others may appreciate simple values such as empty strings. Therefore, it is crucial to create understandable database schema test cases to support the human comprehension of test outcomes that may expedite the process of finding and fixing faults [24].

Another challenge is that a test suite can include many automatically generated tests. Thus, these tests take a long time to execute when changing the schema-under-test (i.e., regression testing) and limit the ability of human testers to understand them. They also include many and repetitive `INSERT` statements that are inefficient while interacting with a database. Therefore, traditional test suite reduction methods (e.g., [25, 26, 27]) can be utilised to address this issue with discarding test cases that cover the same requirements. However, such traditional techniques might not be well-suited to reducing test suites for database schemas.

This thesis addresses these challenges with many empirical evaluations. This includes evaluating a new test data generator for its effectiveness, efficiency, and fault-finding capabilities against AVM. Also, addressing the human costs associated with inspecting and understanding generated tests. Finally, the creation of a novel approach that is superior at reducing database schema tests compared to traditional reduction methods.

1.3 Thesis Aim and Objectives

This thesis aims to investigate and improve automated test data generation in the context of database schema testing. Thus, this thesis tries to answer a high-level research question on what are the strategies that efficiently generate cost-effective database schema tests? Therefore, this thesis answers this question by addressing the challenges mentioned previously with the following main objectives:

- To empirically evaluate the effectiveness and efficiency of a domain-specific test data generator against the state-of-the-art search-based technique.
- To perform a human study to find understandability factors of automatically generated SQL tests.
- To empirically evaluate and improve traditional test suite reduction methods in the context of database schema testing.
- To perform a human study to identify testers' inspection efforts with differently reduced test suites.

1.4 Contributions of this Thesis

This section outlines the five main contributions of this thesis. The first two correspond to new test data generators, and the last three contributions are based on the human oracle cost associated with inspecting generated tests. The produced contributions are according to the above motivations, and are as follows:

C1: An Empirical evaluation of a domain-specific test data generator – An empirical investigation determining the efficiency and effectiveness of a new domain-specific test data generator compared to the state-of-the-art (Chapter 3).

C2: An Empirical evaluation of a hybridised test data generator – An empirical investigation on the effectiveness of hybridising both the domain-specific test data generator and a search-based technique (Chapter 3).

C3: Identifying understandability factors for database schema tests – The creation of readable test data generator variants and the determination of the understandability factors within integrity constraint test data using a human study (Chapter 4).

C4: Evaluating the efficiency of a new reduction technique – The creation of a domain-specific test suite reduction technique that merges tests and an empirical investigation of its efficiency and effectiveness compared to general-purpose reduction techniques (Chapter 5).

C5: Identifying testers' inspection efforts with differently reduced test suites – A human study to identify testers' inspection efforts of either reduced tests (i.e., short tests) or merged tests (i.e., long, yet equivalent tests) (Chapter 6).

1.5 Thesis Structure

This thesis is structured as follows:

Chapter 2: "Literature Review" starts with an overview of software testing, the state-of-the-art test data generation, and test evaluation. The chapter then reviews database testing literature that includes testing of DBMSs, database interactions, queries, and schemas. Finally, the chapter identifies research gaps in automated test data generation with reviewing related work on test data generation inefficiencies, test comprehension, and test suite reduction methods.

Chapter 3: "DOMINO: A Fast and Effective Test Data Generator" presents a new technique that incorporates domain-specific operators into a random generator to generate test data, called DOMINO (DOMain-specific approach to INtegrity cOnstraint test data generation). The chapter empirically evaluates DOMINO against the state-of-the-art technique comparing test data generation effectiveness, efficiency, and fault-finding capabilities. The results and analysis of DOMINO's generated test data have directed the creation of a new hybridised technique that was also empirically evaluated against the original technique.

Chapter 4: "What Factors Make SQL Test Cases Understandable For Testers?" presents new variants of previously studied test data generators to produce more readable tests. These generators are then used in a human study to evaluate the test comprehension of inspecting automatically generated tests. The human study aims to find understandability factors of differently generated test data for database schemas using both quantitative and qualitative studies. Thus, the results will suggest future guidelines for SQL testing understandability that might help future test data generators.

Chapter 5: "STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases" presents a novel approach to test suite reduction called "STICCER", which stands for "**S**chema **T**est **I**ntegrity **C**onstraints **C**ombination for **E**fficient **R**eduction". The technique discards redundant tests using a greedy algorithm while also merging them. It aims to provide test suites with decreased database interactions and restarts, resulting in faster test suite executions and mutation analysis. Therefore, the chapter will empirically evaluate STICCER against three general-purpose test suite techniques comparing their effectiveness and efficiency of reduction, and reduced test suite impact on fault detection.

Chapter 6: "Can Human Testers Effectively Inspect Reduced Versions of Automatically Generated SQL Test Suites?" presents a variant of STICCER to reduce test suites further. The chapter then presents an empirical investigation comparing the new variant technique to the original STICCER technique. Subsequently, the chapter presents a human study to evaluate testers' inspection efforts with test suites reduced using two different reduction techniques: only reduced tests (i.e., short tests) and merged tests (i.e., equivalent, yet fewer long tests). The human study results will help testers in the future to select the best reduction technique that produces easy-to-inspect test suites.

Chapter 7: "Conclusion and Future Work" includes an overall thesis conclusion with summaries of each chapter's contributions, limitations, and recommendations for future research work.

Chapter 2

Literature Review

2.1 Introduction

Software development is considered to be a craft because it requires developers to use their eyes, hands, tools, and materials to build software systems. This craftsmanship can introduce faults into the newly developed software system [28], this can be due to inadvertently omitting, adding, or changing code. With further growth of software code, new faults can be introduced affecting the previously implemented features. For example, some simple mistakes can manifest in minor software crashes that can irritate users, others can cause a loss of an estimated \$500 million, for example, the explosion of Ariane 5 rocket after 40 seconds of launching [29]. Hence, software developers can use testing methods to ensure the implementation correctness. Testing increases the confidence of delivering a highly correct software system [30, 31]. Therefore, software testing is a crucial part of any software development life cycle.

Databases are an integral part of most software applications [32]. They store and manage the software's data, and they are particularly neglected for testing because they are assumed to be correct. Despite that databases are subject to modifications throughout an application's lifetime which can cause failures [33, 34, 35, 22]. Therefore, industry experts recommend testing databases rigorously [13, 34, 33]. This motivated researchers to investigate and create testing methods for databases to be used throughout the software development life cycle.

Software testing can consume large amounts of development time if done manually, leading most software projects to dedicating 50% of the cost towards the testing phases [31, 36]. Motivating researchers to create methods to automate testing by generating test data for both software code and databases. Thus, helping to increase the system correctness and lower the cost of writing tests.

The structure of the literature review is as follows: The first section presents an overview of software testing concepts, the state-of-the-art automated testing methods, and the evaluation of automated tests. The second section reviews relational databases basics, database testing methods, and the state-of-the-art test automation techniques. Finally, a summary of research gaps in automated relational database testing.

2.2 Software Testing

This section will introduce the basics of software testing in general. Then introducing the software testing automation methods, such as Search-Based Software Testing (SBST).

2.2.1 Overview

Amman and Offutt have emphasised the importance of testing within the software development cycle [30]. They reported that software artefacts, such as source code requires developers to test any given requirements for quality assurance purposes and ensuring the system's correctness. That is, any software artefact should work according to the given requirements. However, testing has a limitation of only "show[ing] the presence of bugs, but never to show their absence" as stated by Dijkstra [37]. A failure is defined as an incorrect behaviour of a requirement which caused by a fault. Therefore, a fault is the manifestation of an incorrect internal state (i.e., a fault within the code). Hence, testing is the process of discovering these faults.

Software testing has many levels that are categorised by the location (or grouping) of tests within the software, such as *unit*, *integration*, and *system* testing. Unit testing requires testing each function within a system. For instance, in the context of Object-Oriented Programming (OOP), unit tests are performed on each class and its methods. Integration testing checks the interaction, or integration, between components (e.g., different classes). System testing ensures that the full software system runs correctly with all the user's outlined requirements (i.e., technical, functional, and business requirements).

Software testing methods can be categorised as conceptional views of a box (i.e., the software artefact), such as the *White-box*, *Black-box*, and *Grey-box* approaches. That is, if testers have a view of internal structures while testing (i.e., the software code), then this is considered a white-box (i.e., transparent) approach. However, if testers have no knowledge of the internal structures, then this is considered a black-box approach (e.g., like normal users). The combination of black and white box approaches is considered as a grey-box. For instance, a tester exercises the software with inputs and expected output (i.e., black-box), subsequently reviewing the functions that were executed (i.e., white-box).

Software testing can have other criteria that are non-functional. That is, testing how well the system behaves rather than to only what it does. For example, testing the performance of a system (response times), security concerns, or its usability and accessibility.

Most software test data generation methods require the knowledge of software internals to exercise the software with test data. These methods assume that the developers have knowledge of the internal structures of a program to evaluate it with the generated test data. Henceforth, this literature review will focus on white-box testing as automated methods

2.2.2 White-box Software Testing

A white-box testing approach is usually applied at the unit testing level. A unit test includes inputs and expected outputs to exercise a piece of code. Unit tests are also referred to as test cases because they test a scenario (i.e., the code or test requirement) with a given input. Therefore, test cases can be grouped a test suite, depending on covered code or test requirement.

The construction of test requirements can be based on a *code coverage criterion* in white-box testing. Such that each test case can cover part of the software code. Some basic code coverage criteria include the following:

Statement Coverage – covering and executing a set of code statements.

Function Coverage – covering and executing a function.

Branch Coverage – covering and executing branches (i.e., control structure, such as `IF` statements).

Logical Coverage – covering and executing boolean expressions that must evaluate to either true or false.

Modified Condition/Decision Coverage (MCDC) this requires the tests to invoke each entry and exit of a program, each condition, and each decision must be invoked at least once with every possible outcome.

A statement coverage criterion is defined as simply covering required code statements by a test case. For example, a test requirement state that each line statement in the program must be covered/exercised at least once. The function coverage criterion requires a test to cover a one or multiple functions in a system. For example, exercising each function with inputs and expected outputs.

```
if (x > y) {
  z = 1;
} else {
  z = 0;
}
```

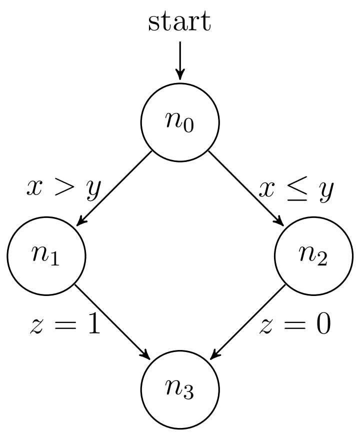


Figure 2.1: Control Flow Graph code fragment example.

A branch coverage criterion uses the structure of a system to create test cases. This structure is observed from the program abstraction graph that represents both the control and data flow. Thus, this criterion requires covering each part of the graph. For instance, Figure 2.1 includes an `IF` statement that can be represented in a graph. The graph includes *nodes* and *edges*, where a node represents a function, source code lines, or a condition (i.e., a control). An edge represents the entry or exit of each node. Therefore, the branch coverage criterion requires tests to cover each node and its edges.

The logical coverage criteria test the logical expressions within a system. Such expressions can be sourced from decisions (i.e., conditions), finite state machines, or requirements. These expressions are represented as *predicates* that returns a boolean result, *true* or *false*. For example, $A < 5 \wedge B > 100$ predicate must be evaluated as true by one test case and as false by another test case. However, a predicate can have multiple clauses joined by logical operators (e.g. $\vee$ denotes OR and $\wedge$ denotes AND). That is, a clause is a predicate without logical operators. For instance, $A < 5 \wedge B > 100$ includes two clauses of $A < 5$ and $B > 100$. This leads the logical coverage to have three criteria: predicate, clause, and the combination coverage.

The predicate coverage criterion requires testing predicates to be evaluated as true or false. Similarly, the clause criterion exercises each clause of the predicate but does not ensure the predicate is fully covered. Therefore, to cover both the clauses and predicates, a tester must use the combinations of both the predicate and clause criteria that is called combination coverage criterion.

The creation of tests with one or multiple criteria manually can be a tedious task. This motivated many researchers to create automated methods to generate test cases with their expected input and output (i.e., test data). The following section will explore these automated testing techniques.

2.2.3 Automated Test Data Generation

Symbolic Execution

Symbolic execution is a method that was first introduced in 1979 [38, 39]. It does not execute the program, but it assigns symbolic values to variables to follow the path of the source code structure. Thus, it generates algebraic expressions from the constraints within the system. These are passed to a constraint solver to satisfy test requirement with applicable values. For example, going back to Figure 2.1, to generate values to hit node 1, the symbolic execution assign value x and y symbols δ and σ, respectively. This leads to generate the following expression $\delta > \sigma$ which needs to be solved by a constraint solver. The constraint solver generates values to satisfy the expression and these values are used for the creation of test cases. Many researchers have surveyed symbolic executions [40, 41, 42, 43].

Random Test Data Generation

Random testing is a technique that executes the program with randomly generated inputs (test data) to observe the program structure [44]. This will likely fail to find *globally* optimal inputs (i.e, solutions/tests that will exercise the target program) because it might search a small segment of the whole input space [43]. Random search is considered unguided test data generator that does not have any defined goal to test a targeted part of the software code. Therefore, research created techniques that devised a goal to test a system using what referred as a fitness function (i.e., goal-oriented search) that evaluates generated inputs (i.e., tests/solutions) on their effectiveness of covering the code. They are considered guided generators that will be reviewed in the following section.

Search-Based Software Testing

Search-Based Software Testing (SBST) is the application of meta-heuristic search techniques to automatically generate test cases for a specific program. It uses techniques such as Genetic Algorithm (GA), Hill Climbing (HC), and Simulated Annealing (SA). SBST research has captured the interest of many researchers in recent years [45]. There are also many surveys discussing and reporting on SBST, like McMinn (2004) [43], Afzal et al. (2009) [46], and Anand et al. (2013) [47]. They have discussed some search techniques in software test data generation and reported on the use of GA, HC, and SA implementations to automatically generating tests that are random in nature.

Hill Climbing Algorithm (HC): The simplest search algorithm that uses a fitness function, as illustrated in algorithm Figure 2.2. Hill Climbing selects a point randomly within the search space, then checks the closest solutions in that space (within the neighbourhood), and if the neighbouring solution is found to have a better fitness score it will then 'move' to this new solution. It will then iterate through this until there is no better solution. The Figure 2.2 states a choice of an ascent strategy, meaning how the search moves around the neighbours. The "steepest" ascent strategy means that all neighbours should be evaluated, and the best solution is selected for the next move. However, a "random" ascent strategy means that neighbours are selected at random to evaluate its fitness, and if the selected neighbour solution is better, it will be selected for the next move. HC generate one solution at time, evaluating the neighbour depending on the ascent strategy and only selecting the best and not accepting worse solutions to rigorously search other solutions. Thus, this will likely lead the technique to select best solution within a small segment of possible solutions (i.e., stuck in local optima), while there are better solution in another segment, as illustrated in Figure 2.3. However, to mitigate this issue the search technique should be restarted multiple times to find the global optima as in Figure 2.4.

```
1 Select a starting solution $s \in S$
2 repeat
3     Select $s' \in N(s)$ such the $obj(s') > obj(s)$
        according to ascent strategy
4     $s \leftarrow s'$
5 until $obj(s) \geq obj(s'), \forall\ s' \in N(s)$;
```

Figure 2.2: Hill Climbing Algorithm — in high level description, solution space S; neighbourhood structure N; and *obj*, the objective function to be maximised, as illustrated by McMinn [43].

The Alternating Variable Method (AVM) [15, 14] is a heuristic that uses a variable search process called *Iterated Pattern Search* [48] which accelerates the HC search method. That was applied successfully in SBST [49, 7]. The AVM tries to explore with moves called "exploratory", in the direction of an improved fitness, where the "pattern" of those movements is considered. The pattern steps iteratively increase in size, if improvements in fitness remain. If the fitness hits a local optimum, the technique will restart the search. This approach was used to automate test data generation in database testing, which will be discussed in the following section.

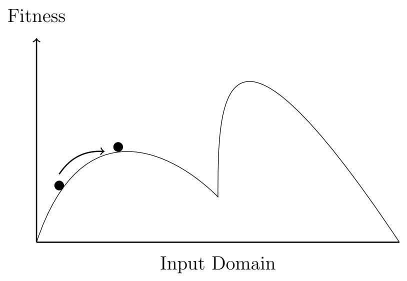


Figure 2.3: Hill Climber searching for a solution and getting stuck in Local Optimum.

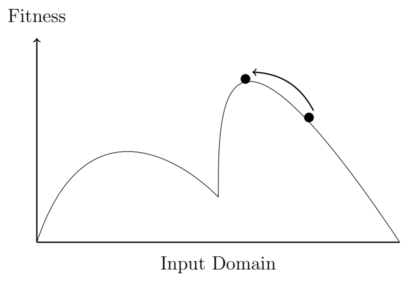


Figure 2.4: Restarting the Hill Climber and finding Global Optimum.

```
int compare(int x, y) {
z = null;
if (x > y) {
z = 1;
} else {
z = 0;
}
return z;
}
```

Figure 2.5: Example of a compare function

Let's consider generating test data that targets a branch to be true, in Figure 2.1, using the AVM search technique. It will first generate random inputs for `x` and `y`, assuming $x = -10 \wedge y = 1$, therefore exercising the false

```
void testCompare() {
int x = 2;
int y = 1
int z = compare(x,y);
assertEqual(z, 1);
}
```

Figure 2.6: An illustrative test example for the `compare` function in Figure 2.5.

branch. To exercise the true branch, the fitness function will be $y - x + K$ applied and minimised to evaluate the generated values (where K is a small positive number, let's say 1), making the current random generated value have a fitness score of 10. So, AVM will start with "exploratory move" on each input variable, either increasing or decreasing the values. Suppose a value can change as ± 1, if the move on x value is decreased the test will still hit the false branch (i.e., $x = -11 \wedge y = 1$) having a worse fitness score of 11. However, if the move increased (i.e., $x = -9 \wedge y = 1$) it will get closer to the target branch and the solution fitness score of 9 is better than the prior score. Incrementally the x value will again create a better fitness score of 8 with $x = -8$. The focus on variable x leads to a "pattern" of better scores that the next move will consider this pattern until the target branch is exercised or the score becomes worse. Making the pattern applied as multiplying the prior change by two ($\pm 1 \times 2$) for each iteration. Going from $x = (+1 \times 2) - 8 = 6$ to $x = (+2 \times 2) - 6 = -2$, then to $x = (+4 \times 2) - 2 = 2$. Therefore, with only 5 moves the solution was found with $x = 2 \wedge y = 1$ and fitness score of -2. On the other hand, if $x = 0 \wedge y = 1$ are the initialised values, AVM will start alternating the x variable to be 1 or -1, which will not hit the true branch. After so, AVM will jump to the y variable and explore it with the move of $x = 1 \wedge y = 0$ hitting the targeted branch.

Simulated Annealing (SA): inspired by annealing of materials with the use of heat temperature to control the search. This is to remove the need to restart the algorithm and overcome the issue of local optima. This is by allowing the algorithm, illustrated in Figure 2.7, to accept poorer solutions with lower fitness score, using a variable called 'temperature'. It initially sets the temperature to 'high', then it will 'cool' down as the algorithm runs. While the temperature variable is high, the algorithm will accept, with high probability, any solutions that has a lower fitness score than the current solution. This will lead the algorithm to jump around optima, early on of the execution. The probability of accepting worse solutions will be high, as the temperature is being reduced to a 'freezing point', therefore leads the algorithm to focus on areas of the space that has an optimum solution. This technique has been used in software testing but not in database testing [50].

```
Select a starting solution $s \in S$
Select an initial temperature $t > 0$
repeat
    $it \leftarrow 0$
    repeat
        Select $s' \in N(s)$ at random
        $\Delta\ e \leftarrow obj(s') - obj(s)$
        if $\Delta\ e < 0$ then
            $s \leftarrow s'$
        else
            Generate random number
             $r, 0 \leq r < 1$
            if $r < e^{-\frac{\delta}{t}}$ then
                $s \leftarrow s'$
            end
        end
        $it \leftarrow it + 1$
    until $it = num\_solns$;
    Decrease $t$ according to cooling schedule
until Stopping Condition Reached;
```

Figure 2.7: Simulated Annealing Algorithm — in high level description, solution space S; neighbourhood structure N; *num_solns*, the number of considered at each temperature level t; and *obj*, the objective function to be minimized, as illustrated by McMinn [43].

Genetic Algorithm (GA): inspired by biological evolution that arises from reproduction, mutation, and survival of the fittest, the GA concept is based on the Darwin's theory of evolution. Like HC and SA, the GA must use a fitness function to rate or score solutions. The algorithm as illustrated in Figure 2.8 starts with a set of chromosomes (i.e., solutions or test data), also called individuals, in a population at random. The population is a set of solutions that are sampled from the search space. The GA then evolves the population with new individuals (solutions), each evolution is called a generation, and each generation evolves from the fittest individuals from the previous population. Over many generations, new solutions are generated and added to the new population until a defined number of evaluations are reached.

The evolution within GA uses three types of operators: *selection*, *crossover* (or recombination), and *mutation*. The selection operator can be used to select the fittest individuals (i.e., the best solutions with the best fitness scores) that are considered during the next evolution. The crossover operator can generate new individuals for each generation by selecting two individuals (i.e., parents), then splitting the individual at a random point and combine them by crossing them over. For example, if the test data of two parents are $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8\}$, then the new crossed over test data are $\{1, 2, 7, 8\}$ and $\{5, 6, 3, 4\}$, if assumed that the split in the middle. These new individuals are usually called an offsprings. The GA also can mutate these individuals using the mutation operator with small changes (e.g., $\{1, 2, 7, 8\} \rightarrow \{1, 2, 7, 10\}$).

```
1 Randomly generate or seed initial population $P$
2 **repeat**
3     Evaluate fitness function of each individual in $P$
4     Select parents from $P$ based on selection mechanism
5     Recombine parents to generate a new offspring
6     Construct a new population $P'$ from parents and offsprings
7     Mutate $P'$
8     $P \leftarrow P'$
9 **until** *Stopping Condition Reached*;
```

Figure 2.8: Genetic Algorithm pseudo-code, as illustrated by McMinn [43].

Global and Local Search. The above algorithms are classified either global or local search techniques. The global search samples solutions from many areas of the search space, however, the local search uses one solution and focuses on one area of the search space (i.e., a neighbourhood). This means a GA technique is considered as a global search technique, meanwhile the other two algorithms, the HC and SA, are local search technique.

Search-Based Software Testing applies these algorithms to cover the program code using a test criterion such as branch coverage. The testing criterion evaluates the generated test data with fitness values. For example, the implementations of EvoSuite [51] and IGUANA [52] both use the aforementioned algorithms. Thus, the SBST research community tries to identify an optimal set of test data that use a coverage criterion, or multiple criteria, in a reasonable time. For example, EvoSuite implements a GA method to generate test data for Java programs and it represents each individual as a test suite (i.e., including test cases). It can generate test suites that cover up to 71% of branches, while random search got 65% coverage all with a budget of one minute [53]. However, there are limitations with using generated test data that must be evaluated/inspected by a tester or a specification for its correctness. This is referred to as the *oracle problem.* If inspected by a human tester, the generated test data can hinder this task as the generated test data can be long and unreadable. Coverage-oriented test data generation can generate inadequate tests; therefore, mutation analysis is used to evaluate such generated test data which is explained in the next section.

2.2.4 Mutation Analysis

Mutation Testing, or mutation analysis, is used to evaluate the quality of existing test cases [54]. This is done by seeding systematic faults in the program code to make it ‘mutated’ (or faulty) code, then executing the test cases against it and review if the fault was detected. If any test case fails then the mutant is **killed** (i.e., detected). Otherwise, mutant is considered **alive** (i.e., undetected). This process determines the quality of the test suite, and the mutation score (i.e., detection effectiveness) is calculated by summing all the killed mutants and divided by the total generated mutants. The advantage of mutation testing that it evaluates the fault-finding capabilities of a test suite, rather than relying only on coverage criteria. That is, coverage criteria only measure the executed code and are not able measure fault detection.

Mutating the code requires *mutation operators* that change a specific code. The most common operators are statement deletion, swapping of boolean expressions or arithmetic operators. For example, replacing a summation sign with a division, or changing of some boolean relations with others such as greater than with less than relations. Mutation testing has been used in relational databases to validate queries and schemas that will be described in the following section.

Listing 2.1 and 2.2, shows a real example of mutation testing. This mutates Line 1 by changing the operator from a "greater than" to a "less than" condition on the mutated code. The test case must kill the mutant which means it must fail as a test, if not, the test has not exercised the actual condition been given.

Listing 2.1: Original code

```
1   if (x > y) {
2   z = 1;
3   } else {
4   z = 0;
5   }
```

Listing 2.2: Mutated Code

```
1   if (x < y) {
2   z = 1;
3   } else {
4   z = 0;
5   }
```

Mutation analysis consume time as there can be many generated mutants (i.e., many versions of the code) and executing test suite on each mutant. For example, many test cases are executed and validated (i.e., regression testing), and mutation analysis is used to assess their fault-finding capabilities. This leads to long waiting times for developers to get the results of failing or passing tests [55]. Also, requiring developers to inspect the tests to reason about their failures, especially with automatically generated test data. Therefore, test suite can be reduced using reduction techniques (i.e., obtaining a representative smaller test suite with fewer test cases) to lower their running times and inspecting the test suites. The following section will review related work in of test data evaluation (i.e., inspecting tests).

2.2.5 Test Data Evaluation (The Oracle)

The test oracle is a process that identifies the correct behaviour of the System Under Test [56]. For example, the inspection of the automatically generated inputs with respect to the expected behaviour. Therefore, test oracles can be constructed and used to inspect the test data from many sources, such as specifications or program executions. The oracle construction has difficulties because the lack of formal specifications that creates a higher oracle cost [57], or the expensive computations to execute the system under tests. Therefore, generated test cases usually do not have an oracle making this known as the "oracle problem" as illustrated in Figure 2.9. Thus, when no specifications are found, the automatically generated tests are usually inspected/evaluated by humans (i.e., testers or developers) manually, however, this a time-consuming task. Barr et al. [56] have discussed many types of oracles extensively, such as *Specified*, *Derived* and *Human* oracles which will be reviewed in the following sections.

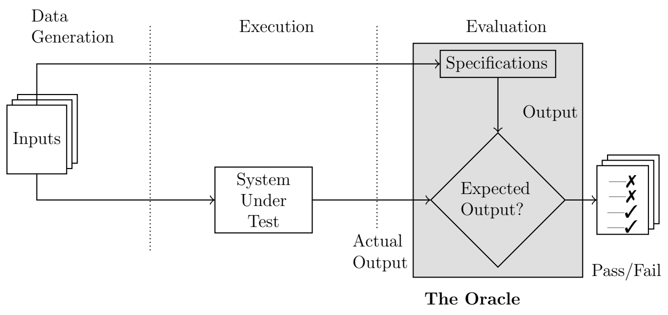


Figure 2.9: An illustration of the oracle problem.

Specified Test Oracle

An oracle information can be sourced from the specification, hence the name Specified Test Oracle. The sources can be specifications, models, finite state machines, and many more. Specifications are documentations of how the system should behave, and they can be informal or formal. Many software systems have informal specification that rely on human natural language [57]. Whereas formal specifications are methods of documenting exactly how the system should behave using mathematical based techniques. They can be written using notations such as Z language [58], and Abstract Machine Notation (AMN) in the B-Method [59]. These notations can be used to generate test oracles to ensure that the system under test behaves correctly. However, using specifications can be limiting as they are lacking within most systems, too abstract, and might be infeasible to use. Others experimented with finite state machines to construct test oracles [60]. While some used assertions within the program code to create assertion-based test oracles [61, 62].

Derived Test Oracle

Deriving the oracle information from documentation or system executions can be categorised as Derived Test Oracle. That are generated when specifications are not available, and they can be pseudo-oracles, metamorphic relations, invariant detection, and many more.

Documentation can provide information to construct test oracles such as sequencing informal text to formal specifications [63]. Another test oracle can be sourced from a version of the system that was written with a different programming language and by another team where the original specifications did not change. This version can be referred as a pseudo oracle that can be executed in parallel with original system using the same test data to derive a test oracle. If the system result is equal, then the original program is considered valid, otherwise it indicates presence of faults in the program or requirements [64]. Therefore, multiple versions of the system can be generated automatically using genetic programming methods [65] or the use of testability transformations [66].

Program invariants (i.e., constraints that always hold true) can be used to derive test oracles [67]. They can be automatically detected using a tool called Daikon [68] that executes the program with test data. Detected invariants can show program behaviours and are used to check the program correctness (i.e., deriving the test oracle).

Derived test oracles are computationally expensive as they require the analysis of one or many sources that infer an oracle. Importantly, inferring derived test oracles can be inaccurate and include many false positives [56].

Human Test Oracle

The effort required by a human acting as an oracle for a test suite — that is, understanding each test case and its outcomes, reasoning about whether a test should pass or fail and whether the observed behaviour is correct or otherwise — is referred to as the "human oracle cost" [56]. Human oracle costs are either quantitative or qualitative. It is possible to decrease the quantitative costs by, for instance, reducing the number of tests in a suite or the length of the individual tests [25, 26, 27]. Strategies to reduce the qualitative costs often involve modifying the test data generators so that they create values that are more meaningful to human testers [69, 70, 24, 71].

Automated test data generation can be helpful to lower the cost of written tests. However, it comes with a cost of manually inspecting test cases. This oracle problem was not tackled in database schema testing as it will be shown in future sections. The following section will review relational databases and their testing methods.

2.3 Database Management Systems (DBMS)

Database Management Systems (DBMS) allow users to organise, store, retrieve, and modify data within a database instance [72]. Organisations consider their databases as the most valuable asset and they are the backbone of most software systems [73]. Databases can be *relational* or *non-relational.* These categories can be considered too generalised because the relations between data can be applied to non-relational databases, such as graph databases. However, Sint et al. [74] categorised databases depending on the methodology of storing the data that can be structured, semi-structured, or non-structured. Structured data layout can be applied in advance using a *schema.* A semi-structured database, known as schema-less, does not require a layout implemented in advance. However, some semi-structured databases can have the option to create such layout (i.e., semi-layout), which are called schema-optional. This section focuses only on relational databases (i.e., structured databases) because many testing methods have been created for them.

2.3.1 Relational Databases Basics

Relational databases enable users (or database engineers) to create a collection of organised data using tables, columns, data types, and constraints. That is, designing a schema in advance which has a set of rules that define the structure of data in a database [72].The concepts of Relational Database Management Systems (RDBMSs) introduced by Codd [75] are based on set theory and have two-dimensional tables with rows and columns. RDBMSs require schemas to navigate the databases. A schema defines the tables and their structure with relationships. Each table has columns that store a piece of data and can impose restrictions on data types, value uniqueness, and if they can be nullable. RDBMSs have a feature of using a high declarative language interface to interact with data, called Structured Query Language (SQL) which can create, access and manipulate data.

SQL can be explained with a simple teacher-course database example with the following set of requirements: (1) A teacher record must have a unique identifier (ID), a full name, and date of birth (dob) and the teacher must be over 18 years old; (2) A course record must also have a unique ID, name, and starting date; (3) Each course must have one teacher. These requirements can be designed into a schema using SQL `CREATE TABLE` commands. Thus, the schema should include two tables, `teacher` and `course`, as illustrated in Figure 2.10. Each table must have a set of columns; hence, the teacher table will have `teacher_id`, `fullname`, and a `dob`. Each column must have a data type such as `int` for a numeric and `varchar` for characters. As a requirement, the teacher ID must be unique, and the date of birth must be over 18 years old. Therefore, the `UNIQUE` constraint and a `CHECK` constraint to defend and preserve the consistency and coherency of the required data. A `PRIMARY KEY` constraint is also used to make the identifier unique, like the `UNIQUE` constraint. For this example's sake both the `PRIMARY KEY` and `UNIQUE` will be used together. Like the `teacher` table, the `course` table is built using data types and integrity constraints. However, the last requirements require each course to have one teacher, thus the `FOREIGN KEY` constraint was used in the `course` table to be linked with a teacher ID.

```
CREATE TABLE teacher (
  teacher_id INT PRIMARY KEY,
  fullname VARCHAR(255) NOT NULL,
  dob DATE,
  CHECK (dob < (‘now’, ‘-18’)
);

CREATE TABLE course (
  course_id INT PRIMARY KEY UNIQUE,
  course_name VARCHAR(255),
  starting_date DATE,
  teacher_id int
  FOREIGN KEY (teacher_id) REFERENCES
    teacher(teacher_id)
);
```

Figure 2.10: An example of a course schema to illustrate the use of creating schemas using SQL `CREATE TABLE` command.

The SQL gives a rich number of commands to add, retrieve, modify and remove records. In Figure 2.11 a record is added in the teacher table using the `INSERT` command (Line 1), retrieve all rows in the teacher table can be done with the `SELECT` command in Line 2. However, retrieving one record from the teacher table where the ID is 1 can be done with the `SELECT` and `WHERE` commands (Line 3). Updating a record (i.e., modifying a piece of data), Line 4 uses the `UPDATE` command to change the date of birth for a teacher record that has an ID of 1. Deleting a record can be done using a `DELETE` command, Line 5. SQL also can create more complex queries using `JOIN`s and `GROUP BY`.

```
1)  INSERT INTO teacher (teacher_id, fullname, dob) VALUES (1,'John',23-10-1990);
2)  SELECT * FROM teacher;
3)  SELECT * FROM teacher WHERE teacher_id = 1;
4)  UPDATE teacher SET dob=11-11-1911 WHERE teacher_id = 1;
5)  DELETE FROM teacher WHERE teacher_id = 1;
```

Figure 2.11: Demonstrating SQL queries

In relational databases, there has been major research development regarding software testing and the programs that are reliant on them. If the DBMS or even the database has any faults then the application that relies upon it would be error prone, which leads to either data loss or inconsistencies of the data stored [76, 20]. Therefore, the following sections will cover related work on testing of relational databases such as DBMSs, database interactions, queries, and the focus of this thesis testing schemas.

2.3.2 DBMS Testing

DBMSs are important to manage data and therefore must be tested. Testing ensures that users retrieve the expected query results. This motivated many researchers to create testing methods for relational DBMSs [77, 78, 79]. In order to test a DBMS, testers must populate the database with data and produce a set of queries with their expected results. However, with a given schema, there is an enormous number of queries that can be created which target different parts of the DBMS. Automatically generating all possible queries according to a given criterion is considered as a 'sampling' approach [77, 78, 79]. However, producing queries systematically to test certain parts of the DBMS is considered as 'systematic' approach [80, 81, 82, 83]. These approaches require the database to be populated with data to exercise the queries with either a given a set of queries or a schema. Therefore, this section will discuss the techniques relating query and data generation.

Query Sampling Approaches

There are many test automation techniques that generate test queries for testing DBMSs such as random techniques, constraint solvers, and search-based techniques. Slutz [77] was the first to introduce an automated tool called RAGS (Random Generation of SQL) for relational DBMSs that randomly generates queries for enterprise systems. This tool was used to generate random queries based on a given schema to evaluate the database loss of connection, compiler errors, execution errors, and system crashes. Therefore, successful queries that crash the system will be saved for regression testing. RAGS work on a pre-populated database and then generates queries by traversing the SQL statement tree and randomly adding or modifying elements on the statements that create more complex `SELECT` queries. The addition or modification can be on the columns, tables, `WHERE` conditions, and group clauses. The results of running the queries on different versions of the DBMS are then compared. If they are different, then the assumption that one of them is faulty. This tool can generate up to three million queries per hour and helps find bugs within a DBMS.

RAGS was improved by Bati et al. [78] using a genetic algorithm (GA) to generate `SELECT` statements that cover a goal. With a pool of queries, the algorithm runs by ranking each statement depending on the goal. Thus, using RAGS to mutate the queries and generate new ones for the next evaluation. The fitness function evaluates the queries with the number of paths discovered within the DBMS by checking the logs and execution time. Thus, trying to cover most of the system. The new GA technique was compared to the original technique (i.e., RAGS) and achieved around 29% more coverage with both techniques having the same time budget.

QGEN is a technique that was proposed by Poess and Stephens [79] to generate queries with a given template language. That is, creating template of variables. The technique then generate data randomly with a predicable distribution set. For example, generating random years in a query with an assigned to a variable range between 1900 and 2000. The technique was not rigorously evaluated but it can be used to test the performance of DBMSs.

Query Systematic Approaches

This approach aims to test the DBMS in a structured way. That is, using a model of the system's constraints and constraints solvers to generate queries [84]. Therefore, the constraint solver must be given a described schema in Alloy language format to generate all possible valid `SELECT` queries. The queries include many elements such as grouping or joins or even aggregation operators. These queries are meant to cover all elements of the schema and test against the DBMS. However, the technique can generate many queries depending on the given schema. For example, Khalek and Khurshid [84] generated over 27,000 queries in around two minutes for a simple two table schema and the DBMS can take a long time to execute such queries.

Automatically populating DBMSs

Testing a DBMS requires the database to be populated with rows for the query to return results. Thus, it needs to be populated either with prior knowledge of the queries that will be executed, or with data that exhibit the schema design. The former is categorised as 'query-oriented' approach [80, 81, 82, 83] and the latter is 'schema-oriented' approach [85, 86].

Query-oriented approaches can help in testing targeted parts of the DBMS by constructing the database with specific data to satisfy results for queries. There are tools that generate the test data automatically such as QAGen [80], DataSynth [81, 82], and ADUSA [83]. QAGen generates data with user defined constraints of the schema and the queries. For example, the user can define the number of rows for each part of a given query using cardinality constraints. That is, the `course` table is 100 rows to be generated for `SELECT` queries whereas `JOIN`s and other operations can have their own sizes. QAGen will then generate and guarantee the number of rows retrieved by a given query and schema. This generation will help testers to get the desired data of a query and help to test a range of DBMS tasks such as memory managers and query optimisers. The study showed that QAGen produced 10 megabytes of test data for each query in $\sim$14 minutes and up to 1 gigabyte in $\sim$27 hours.

DataSynth [81, 82] generate test data similar to QAGen however it produces data simultaneously for many cardinality constraints rather than generating data for each query. Using integer linear programming compared to QAGen which uses a constraint solver, DataSynth was more efficient to generate test data compared to QAGen. For instance, DataSynth solver took under 5 seconds compared to QAGen solver that took over 10 minutes for the same query that required 1 gigabyte of test data.

The Alloy language was used to model queries and schemas to generate test data. An approach Khalek et al. [83] created to utilise the Alloy Analyser for generating test data, called ADUSA. If possible, the analyser tries to satisfy the given constraints with data, guaranteeing returned data for the query. The analyser results can be compared to the DBMS results that removes the need of a test oracle or other DBMSs for comparisons. The empirical evaluation of ADUSA showed 1,000 test databases were generated in under 78 seconds for a five-table schema that included `FOREIGN KEY`s. However, the queries are simple and only had a `where` predicate. If more rows are required, the generation time will take significantly longer. Their results showed that ADUSA successfully detected bugs in commercial and open source DBMSs.

Producing test data with prior knowledge of queries and schemas can be helpful to control the tests and the amount of results returned by the DBMS. However, if testers want to test the DBMS on its structure and relationships of the schema without prior knowledge of the queries, they can use the schema-oriented approach. Therefore, generating large amount of data for a given schema. For instance, Bruno and Chaudhuri [85] created a technique that annotates the schema to generate data, referred to as flexible database generation. The technique employed statistical distribution to generate test data as normal or uniformed data. It also ensures that the data does not violate the integrity constraints defined in the schema and the relations between tables are upheld. The tester can annotate the schema and select a statistical distribution with a size variable (i.e., the number of rows) and it will automatically generate data. The size variable can be used to test the DBMS performance. This technique showed it can generate $\sim$1GB of data around 13 minutes, translating to 10.3 million rows.

Houkjær et al. [86] created a schema-oriented approach that utilises a graph model (i.e., edges and nodes) to represent a schema. The node in the graph represents table columns, data types, and the number of rows required. The edges represent the relations between nodes (i.e., `FOREIGN KEY`s). The graph then is used to generate data and populate the database by traversing the graph, upholding the relationships of the schema. This technique generated 1GB of data within 10 minutes. Both the graph model and annotation methods can be useful to generate data for a given schema. However, the data does not guarantee queries to have returned results.

All aforementioned studies focus on testing the DBMS, the large system that manage databases. However, the work in this thesis focus on testing database schemas. Nonetheless, the following section explore the related research with database interaction and query testing.

2.3.3 Database Interaction Testing

Testing DBMSs does not ensure the correctness of the application and database interaction. Thus, testing the application that uses the database to store the data is more challenging. That is because the application state can be influenced by the data within a database. For example, queries embedded within the application can be faulty. This motivated some researchers [87, 88, 89, 90] to test the interactions between databases and applications which can be called "database application testing". For example, fault-based testing was used by Chan et al. [88] with mutating the database entity-relationship model (i.e., a schema model). They introduced seven mutation operators that change the model with semantic mutants (i.e., that change the meaning rather than the code). Therefore, they can be tested against the embedded SQL queries. However, this concept has not been prototyped and empirically evaluated.

The use of data flows between a program and the database interaction can be tested using criteria introduced by Kapfhammer and Soffa [87]. The testing criteria were based on def-use (i.e., variable definition and its uses within a program) that ensures test cases capture the interaction between the program and the data store. That is, observing the changes in entered values within a program and the storing of these values. They showed that two applications had test suites that significantly overlooked the database interactions when using def-use testing methods. Therefore, their criteria can be used to test database applications and test data generation based on these criteria.

Many others have created their own coverage criteria to test database applications [91, 92]. Such as using a command-form (i.e., SQL commands issued to the database by the application) criteria [92]. Another proposed a structural coverage criterion that requires all the conditions within an SQL query (i.e., `FROM`, `WHERE`, and `JOIN`) to be tested [91].

Chays et al. proposed a framework called AGENDA that aims to generate test cases that check the program queries, retrieved results, and the database states [89]. That is, parsing the schema, populating the database with satisfying the integrity constraints. Then, generating test data is used and entered into the application. Therefore, the test data and the results are checked against the database. Following this, the technique generates a report to assist the tester to inspect the queries embedded for any issues. For example, undesired values can enter the database which the tester can see as a fault within the queries embedded in the application. The AGENDA framework was then extended with query transactions and was empirically evaluated [93]. They experimented with three database applications that were seeded with 15 faults. The results showed that AGENDA detected more than half the faults (i.e., 52% detection) in less than a minute.

Testing database applications might require the database to reset to the initial state. For example, a test case might modify the database that can affect other test cases results. Especially with test cases that include many test data, requiring the database reset its state. This issue is considered non-trivial and motivated Haftmann et al. [94] to propose a set of algorithms that orders the test case executions without the need to reset the database state and with the aim to have correct results. Their empirical results showed that their technique significantly reduced a large set of databases using a thousand test cases. However, their techniques resulted of some false positives because the database state can be empty, and the tests passes.

Arcuri and Galeotti [95] improved their EvoMaster framework that generates test data for web services (i.e., REST and SOAP) by ensuring that the database state is correct and populated. EvoMaster generates tests at system level for web APIs and utilises an evolutionary algorithm (e.g., a GA). Testing an API sometimes requite interacting with database by creating, retrieving, and updating data. Therefore, they proposed a heuristic that monitors the database interactions and tries to maintain and populate the database with test data. Their technique generates a list of `INSERT`s that populates that databases with random values. However, when values conflict (i.e., violation of a `PRIMARY KEY`), there is a repair method that tries to fix such conflicts with randomisation of values. The `FOREIGN KEY` values are fixed with the use of auto-incremental values of the prior `INSERT`s. Their empirical evaluation showed that generating test data for the database increased their coverage by 18.7%. Therefore, the consistency of the database state helped with testing web service APIs. This work relates to database testing with exercising some of the schema's behaviour. However, it does not fully test schema's with defined criteria as this thesis. The use of "fix" technique is somewhat related to the test data generator that will be evaluated in the following chapter.

2.3.4 Query Testing

Queries are a major part of any system, and they are required to be tested to ensure that they retrieve the correct results. Thus, testing constructed queries is important and motivated many to create testing methods for them. For example, Tuya et al. [90] developed test coverage criteria called SQLFpc based on a program modified condition/decision criteria. It considers different semantics of SQL syntax and schema constraints. The criteria require testing `SELECT` statements, considering sub-queries such as joining and grouping. For example, if `SELECT * FROM a WHERE x` $\geq$ `10` then the test requirement would be testing `x > 10`, `x = 10`, `x < 10`, and testing it with `x = NULL`. Therefore, exercising each part of the `WHERE` clause as true and false, and any other part of the query such as `join` must be tested too.

Tuya et al. [90] test coverage criteria were automated by Riva et al. [96] with the Alloy language. That is, generating test data within the database to satisfy the test requirements of SQLFpc coverage criteria. The tester must create a set of coverage rules that cover the SQLFpc criteria and expressed in the Alloy relational language. This is then passed into the tool with an encoded Alloy schema to be solved with generated test data. Thus, each rule will have generated test data that are stored within the database to return results for test query. This approach was empirically evaluated based on coverage and mutation analysis that was introduced by Tuya et al. [97]. The mutation analysis mutates the clauses of the `SELECT` SQL command (e.g., the above statement can be mutated as `SELECT * FROM a WHERE (x-1)` $\geq$ `10`). Using a case study that has production data and many queries, they evaluated the automatically generated test data with the production data on the case study's queries and mutation analysis. The results showed the SQLFpc tool got 86.67% coverage compared to 57.33% of the production data. The mutation score of their test data was 84.13% compared to 66.54% of the production data. The tool also generated 139 rows, which was significantly fewer than the production database that included 139,259 rows. Thus, making the tool more efficient to test queries with useful test data.

Production data can be copied and used for testing. However, the large amount of data can be inefficient and sometimes ineffective [96]. Therefore, Tuya et al. [98] created a method that reduces the production data to ensure efficient and effective testing. Using a greedy algorithm to reduce the test data while maintaining the queries' coverage. They empirically evaluated their technique with four case studies that contain rows ranging from 137,490 to 86,805. The results showed that the greedy algorithm reduced the data from 137,490 to 194 rows with an increase in coverage of 1.5% for one case study. The fault-finding capability (i.e., mutation analysis scores) only declined by 0.3% for the same case study. All of their results indicate that their technique is scalable on production data and will help with efficient regression testing.

Castelein et al. [99] applied search-based testing methods rather than constraint solvers. Because the constraint solvers did not deal well with complex quires such as `JOIN`s. Therefore, they implemented random search, biased random search (i.e., a random search that uses pool of constants mined from queries), and genetic algorithms (GAs) in a tool called EvoSQL. Utilising SQLFpc coverage criteria, they created their own fitness functions. Thus, EvoSQL generates test data for the database with a given SQL query, the database schema, and coverage requirements. They empirically evaluated their approaches with extracted queries from four software systems, totalling 2,106 queries. Their results show that the random search obtained 6.5% coverage of all the queries. The GA obtained 98.6% coverage between 2-15 seconds. However, the biased random search obtained 90% coverage with a competitive efficiency that generated data faster than the GA when there are low number of coverage targets (e.g., a branch or a statement). Therefore, random search can be beneficial for efficiency in some cases.

In summary, query and database interaction testing can help improve the program quality that rely on a database. That is, correctly retrieving and manipulating stored data. This section shows the need to obtain effective and efficient automated techniques for testing databases. However, ensuring that database is consistent and coherent requires a correct schema. Therefore, the following section and chapters will focus on testing database schemas.

2.3.5 Schema Testing

One of the most important artifices of a relational database is the schema and it is the focus of this thesis. It structures how the data should be stored within a database. It includes integrity constraints that defend and preserve the consistency and coherency of data. For example, they prevent duplicate usernames and negative prices. Therefore, any wrong implementations will lead to unwanted and maybe corrupt data within the database. Schemas are often implied to be correct and are implemented with no tests [33]. Moreover, they are always changing throughout the system's lifetime [34, 35, 22].

Like other software artefacts, integrity constraints are subject to errors of omission and commission [30]. An example of an omission error is a developer forgetting to add a constraint on a column, such as not defining a `UNIQUE` constraint on a `username` column. Conversely, a commission error would be a developer unintentionally adding an integrity constraint, such as a `UNIQUE` constraint on a column representing somebody's first name (these mistakes may happen in combination, as the unique constraint may have been intended for a column representing some distinctly identifiable information, such as an identification number). For these reasons, industry experts recommend thorough testing of integrity constraints [13, 34, 33].

Testing the integrity constraints, an approach that will be used in this thesis, can be accomplished by inserting data within tables exercising each constraint as true (accepted by the database) or false (rejected by the database). Therefore, each integrity constraint can be treated as a predicate and covered similarly to logical coverage of program testing, as described in Section 2.2.2. To guide this process, the current state-of-the-art testing coverage criteria, based on logical coverage, for relational database schemas was developed by McMinn et al. [7]. That is, testing integrity constraints within schemas using any of the nine different coverage criteria for testing schemas. These were categorised into two main coverages, *constraint coverage*, and *column coverage* criteria. Covering constraints are based on basic logical coverage. However, covering a column requires testing them for uniqueness and nullability. The following are the nine coverage criteria:

- Constraint coverage criteria:
    - Acceptance Predicate Coverage.
    - Integrity Constraint Coverage.
    - Active Integrity Constraint Coverage.
    - Condition-Based Active Integrity Constraint Coverage.
    - Clause-Based Active Integrity Constraint Coverage.
- Column coverage criteria:
    - Unique Column Coverage.
    - Null Column Coverage.
    - Active Unique Column Coverage.
    - Active Null Column Coverage.

Constructing tests based on the above coverage criteria sometimes requires the database to be prepared in the correct state and to ensure the test requirement is satisfied. For example, testing a `UNIQUE` constraint requires the database to populated for comparison and to exercise the integrity constraint with a test `INSERT`. Testing also may require a populated table to test another table because of relational constraints (i.e., `FOREIGN KEY`s). That is, each test case, depending on the test requirement, require the database state to be prepared to exercise certain integrity constraints. This is called T-Sufficiency of a test case and formally defined as follows:

Definition 1 (T-Sufficiency)**.** The data within a database d for the schema-under-test s is considered T-Sufficient with respect to some test requirements $tr \in TR$ if: 1) tr is not satisfied by the insertion of an arbitrary row of data into d; and 2) The contents of d do not render tr infeasible.

Acceptance Predicate Coverage (APC) is the simplest criterion that requires two test cases for each table within a schema. These two test cases should exercise the table as true (i.e., test data accepted by the database) and false (i.e., test data rejected by the database) using `INSERT` statements.

```
CREATE TABLE places (
  host TEXT NOT NULL,
  path TEXT NOT NULL,
  title TEXT,
  visit_count INTEGER,
  fav_icon_url TEXT,
  PRIMARY KEY(host, path)
);
```

```
CREATE TABLE cookies (
  id INTEGER PRIMARY KEY NOT NULL,
  name TEXT NOT NULL,
  value TEXT,
  expiry INTEGER,
  last_accessed INTEGER,
  creation_time INTEGER,
  host TEXT,
  path TEXT,
  UNIQUE(name, host, path),
  FOREIGN KEY(host, path) REFERENCES places(host, path),
  CHECK (expiry = 0 OR expiry > last_accessed),
  CHECK (last_accessed >= creation_time),
);
```

Figure 2.12: The *BrowserCookies* relational database schema as illustrated and studied by McMinn et al. [7]

The rejection test must at least violate one of the constraints while an acceptance test must satisfy all the integrity constraints. Therefore, APC is defined as follows:

Criterion 1. Acceptance Predicate Coverage (APC). For each table *tbl* of the schema-under-test s, two test requirements are added to TR: one evaluates to true, and one evaluates to false.

Figure 2.13 shows an example APC with two test cases constructed for the `places` table in of the *BrowserCookies* schema in Figure 2.12. Figure 2.13a illustrates a test case that exercises the table's constraints as true. Conversely, Figure 2.13b illustrates the false (expecting a rejection) test case for the `places` table by violating one of the constraints, in this instance the `PRIMARY KEY`. Both test cases include two `INSERT` statements, the first ensures that the database state is prepared (i.e., T-sufficient) for the second test's `INSERT`. Therefore, the *BrowserCookies* test suite, using this criterion must have four test cases, two test cases per table.

The APC criterion tests each table within a schema but does not exercise a specific integrity constraint. Therefore, Integrity Constraint Coverage (ICC) criterion aims to exercise each internity constraint in the schema as true and false. ICC is defined as follows:

Criterion 2. Integrity Constraint Coverage (ICC). For each integrity con-

```
1) INSERT INTO places(host, path, title, visit_count, fav_icon_url)
   VALUES ('amazon.com', '/login.html', 'Log-in', 0, '')              ✓
2) INSERT INTO places(host, path, title, visit_count, fav_icon_url)
   VALUES ('amazon.com', '/home.html', 'Home', 0, '')                 ✓
```

(a) Test Case 1 exercise the `places` table as true

```
1) INSERT INTO places(host, path, title, visit_count, fav_icon_url)
   VALUES ('amazon.com', '/login.html', 'Log-in', 0, '')              ✓
2) INSERT INTO places(host, path, title, visit_count, fav_icon_url)
   VALUES ('amazon.com', '/login.html', 'Login', 0, '')               ✗
```

(b) Test Case 2 exercise `places` table as false

Figure 2.13: Example of Acceptance Predicate Coverage (APC) test cases for the `places` table in Figure 2.12. The ✓ and ✗ illustrate that the database acceptance or rejection of the `INSERT`, respectively.

straint *ic* of *s*, two test requirements are added to *TR*, one where the *ic* evaluates to true, and one where it evaluates to false.

ICC require 20 test cases for the *BrowserCookies* schema that include ten integrity constraints (i.e., each IC is tested as true and false). For instance, Figure 2.13 also show that the `PRIMARY KEY` constraint is tested with both test cases, one with satisfaction and the other is rejection.

The ICC does not require the satisfaction of other constraints when testing a specific constraint. This lead to weaker test cases with `INSERT`s not focusing on the required integrity constraint. For example, a tester that use ICC will exercise a `PRIMARY KEY` as false (i.e., expected violation from the DBMS) while also include a `NULL` value in a `NOT NULL` defined column, failing both constraints. Therefore, having a weaker test that might detect changes to the `PRIMARY KEY`.

Therefore, all the constraints must be satisfied to ensure the constraint under test is exercised with greater precision. Hence, the Active Integrity Constraint Coverage (AICC) criterion aims to exercise each defined integrity constraint as true and false while all other constraints evaluate as true. The AICC therefore is defined as follows:

Criterion 3. Active Integrity Constraint Coverage (AICC). For each table *tbl* of *s*, let each $ic_i \in IC$ under test evaluate as true and false while other integrity constraints $ic_j \in IC, j \neq i$ evaluates to true. TR contains the following requirements: one where evaluates the ic_i to true, and one where it evaluates to false.

For example, exercising the `PRIMARY KEY` in `cookies` table of the *Browser-Cookies* schema, the `id` column, require AICC tests to satisfy all the integrity constraints defined in the `cookies` table. That is, creating a value for the `name` column rather than a `NULL`, and satisfying all the `CHECK` constraints and `FOREIGN KEY` constraint.

The above criteria only exercise constraints as a true and false while relational databases have another state called "unknown" (i.e., allowing `NULL`s). For example, a nullable column that is involved in a `CHECK` constraint allows `NULL`s. Also, some DBMSs interpret SQL standard differently and allow `NULL`s into `PRIMARY KEY` columns (e.g., SQLite) while other DBMSs disallow such behaviour (e.g., PostgreSQL). Therefore, each integrity constraint should be tested as true, false, and with a `NULL`-condition (i.e., DBMS dependent on the behaviour of `NULL` and its truth value). Hence, the next criterion aims to test such shortfall:

Criterion 4. Condition-Based Active Integrity Constraint Coverage (Clause-AICC). For each table *tbl* of s, let each $ic_i \in IC$ under test evaluate as true, false, with a null condition while other integrity constraints $ic_j \in IC, j \neq i$ evaluates to true. TR contains the following requirements: one where evaluates the ic_i to true, and one where it evaluates to false, and one with a null-condition.

The CondAICC begin exercising each constraint as true and false similar to AICC. Then it requires a test to exercise the `NULL`-condition. That is exercising the constraint with a `NULL` and evaluating the truth value depending on the DBMS behaviour. For example, in testing a `PRIMARY KEY` null-condition and the DBMS is PostgreSQL, then the `NULL` value will be evaluated as false. Conversely, with SQLite the `NULL` value will be evaluated as true.

Integrity constraints may include multiple clauses that need to be exercised in isolation such as composite keys and multi-clause `CHECK` constraints. Thus, the Clause-Based Active Integrity Constraint Coverage (ClauseAICC) criterion aims to exercise each clause of such constraints as true, false, and with a `NULL`-condition. It is, therefore, defined as follows:

Criterion 5. Clause-Based Active Integrity Constraint Coverage (Clause-AICC). For each table *tbl* of s, each $ic_i \in IC$ under test evaluate as true, false, with a null condition while other integrity constraints $ic_j \in IC, j \neq i$ evaluates to true. Let c be the set of atomic clauses of ic_i, that is the joined sub-expressions through the logical connectives $\wedge$ and $\vee$. Let each $c_k \in C$ evaluate as true, false, and null condition while the other clauses $c_m \in C, m \neq k$ evaluate as true. TR contains the following requirement: one where evaluates the c_k to true, and one where it evaluates to false, and one with a null condition.

For example, the `CHECK (expiry = 0 OR expiry > last_accessed)` in the `cookies` has two clauses that should be exercised as true, false, and `NULL`-condition. That is, the `expiry = 0` need to evaluate as true while the `last_accessed` value needs to be under zero. As for the violation test `expiry` value can be over zero while `last_accessed` maintain the same value, and similarly with the `NULL` value. Afterword, the `expiry` must be zero while `last_accessed` is exercised with another three test cases similar to the first clause.

Another example is the `UNIQUE(name, host, path)` in the same table that require tests to exercise each of the three columns as true, false, and `NULL` while the rest of columns evaluate as true. This composite `UNIQUE` key requires the tester to create seven tests. Three of which exercise each column with a null condition, another three with a true condition, and one as false (i.e., to reject a composite key, all columns must be identical to one existent row in the database).

The aforementioned criteria test defined integrity constraints. However, they do not test for omitted integrity constraints. For instance, a "username" column not being declared as `UNIQUE` or a "name" column is not being declared as `NOT NULL`. Therefore, each of the column in the schema must be tested as a `UNIQUE`, non-`UNIQUE`, and with `NULL` and not-`NULL` values. That is using the Unique Column Coverage (UCC) criterion and Null Column Coverage (NCC) criterion, respectively. Therefore, the formal definitions of these criteria are as follows:

Criterion 6. Unique Column Coverage (UCC). For each table *tbl* of a schema s, let CL be the set of columns. Let *nr* be a new row to be inserted into *tbl* and *er* be the existent row. For each $cl \in CL$, let $ucl \leftarrow \forall er \in tbl : nr(cl) \neq er(cl)$. TR contains two requirements for each *cl*, one in which $ucl = true \land nr(cl) \neq NULL$, and one where $ucl = false \land nr(cl) \neq NULL$.

Criterion 7. Null Column Coverage (NCC). For each table *tbl* of a schema s, let CL be the set of columns. Let *nr* be a new row to be inserted into *tbl*. For each $cl \in CL$, let $nncl \leftarrow nr(cl) \neq NULL$. TR contains two requirements for each *cl*, one in which $nncl = true$, and one where $nncl = false$.

The UCC requires two tests to exercise each unique column in a table, where one test has unique values to be accepted and the other test has non-unique values to be rejected. However, these test requirement do not need the satisfaction of other integrity constraints such as `PRIMARY KEY` constraint. To consider other constraints, the Active Unique Column Coverage (AUCC) criteria must be used, which requires each test to comply with all constraints while exercising the required column. Like UCC, NCC must require two tests that exercise each column with `NULL` and not-`NULL` values. To also consider the other constraints, the Active Null Column Coverage (ANCC) criteria can be used to exercise each column while complying with all constraints. The formal definitions of AUCC and ANCC are as follows:

Criterion 8. Active Unique Column Coverage (AUCC) For each table *tbl* of a schema s, let CL be the set of columns. For each $cl \in CL$, let *nr* be a new row to be inserted into *tbl*, and let $ucl \leftarrow \forall er \in tbl : nr(cl) \neq er(cl)$. Let ic_{aucc} be the columns for *tbl* that does not account for integrity constraints that require *cl* to be individually unique (i.e., `UNIQUE` constraints and `PRIMARY KEY` constraints defined on *cl*). TR contains two requirements for each *cl*, one in which $ucl = true \land nr(cl) \neq NULL \land ic_{aucc} = true$, and one where $ucl = false \land nr(cl) \neq NULL \land ic_{aucc} = true$.

Criterion 9. Active Null Column Coverage (ANCC) For each table *tbl* of a schema s, let CL be the set of columns. For each $cl \in CL$, let *nr* be a new row to be inserted into *tbl*, and let $ancl \leftarrow nr(cl) \neq NULL$. Let ic_{ancc} be the columns for *tbl* that does not account for integrity constraints that require *cl* to be individually `NULL` (i.e., a `NOT NULL` constraint on *cl*; or a `PRIMARY KEY` constraint defined for *cl* only, in the case of a non-SQLite database). TR contains two requirements for each *cl*, one in which $ancl = true \wedge ic_{ancc} = true$, and one where $ancl = false \wedge ic_{ancc} = true$.

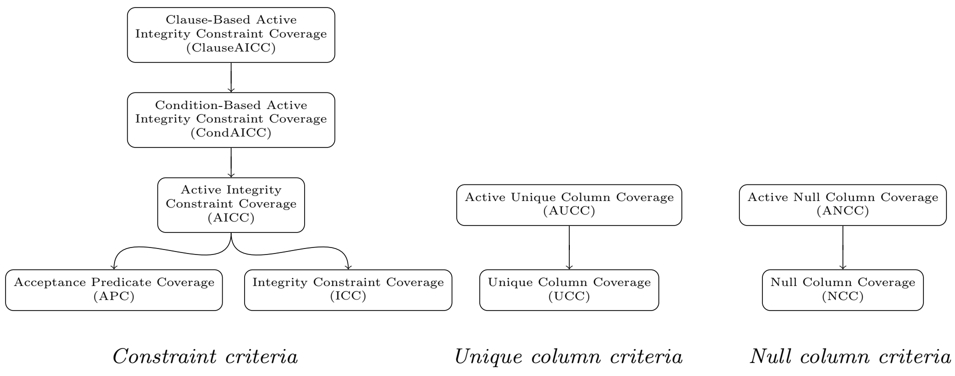

Clause-Based Active Integrity Constraint Coverage (ClauseAICC)
Condition-Based Active Integrity Constraint Coverage (CondAICC)
Active Integrity Constraint Coverage (AICC)
Acceptance Predicate Coverage (APC)
Integrity Constraint Coverage (ICC)
Active Unique Column Coverage (AUCC)
Unique Column Coverage (UCC)
Active Null Column Coverage (ANCC)
Null Column Coverage (NCC)
Constraint criteria
Unique column criteria
Null column criteria

Figure 2.14: The coverage criteria subsumption hierarchy for testing relational database schemas.

All the aforementioned criteria have a hierarchy that one criterion might subsume another. Therefore, in Figure 2.14, the subsumption hierarchy show the strong criterion at the top. Hence, creating tests using ClauseAICC criterion will satisfy the requirements of all the constraint criteria.

These coverage criteria can demand many test requirements to cover the logical predicates. Therefore, they can be generated automatically. For example, to create a ClauseAICC test that exercises the `name` column of the composite `UNIQUE` key in *BrowserCookies* schema as *true*, the following predicates must be satisfied in conjunction with each other and assuming that the database is already populated:

1. The new row (nr) of the `cookies` table must include an `id` column value that is not `NULL` and distinct to existing row (er) `id` value:

   $\mathbf{PK} \leftarrow nr(cookies.id) \neq NULL \wedge \; (\forall er \in nr(cookies.id) \neq er(cookies.id))$

2. The new row must also have a `name` column value and not equal to `NULL`:

   $\mathbf{NL} \leftarrow nr(cookies.name) \neq NULL$

3. The new row of the `cookies` table must have connecting `FOREIGN KEY`s. Thus, the `host` and `path` must be equal to the `host` and `path` of the `places` table or either/both columns equal to `NULL`:

   $\boxed{\textbf{FK} \leftarrow (\forall er \in nr(cookies.host) = er(places.host) \wedge nr(cookies.path) = er(places.path)) \vee nr(host) = NULL \vee nr(path) = NULL}$

4. The new row of the `cookies` table must have the `expiry` column value equal to zero or greater than the `last_access` or either/both values equal to `NULL`:

   $\boxed{\textbf{CH}_1 \leftarrow (nr(expiry) = 0 \vee nr(expiry) > nr(last\_access) = unknown) \vee (nr(expiry) = 0 \vee nr(cookies.expiry) > nr(last\_access) = true)}$

5. The new row of the `cookies` must have the `last_accessed` column value greater or equal to `creation_time`, or either of the columns equal to `NULL`:

   $\boxed{\textbf{CH}_2 \leftarrow (nr(last\_access) \geq nr(creation\_time) = unknown) \vee (nr(last\_access) \geq nr(creation\_time) = true)}$

6. The new row of the `cookies` must have equal `host` and `path` values to existent rows in the `cookies` table. The new row must have a distinct value for the `name` column and all the three columns must be not equal to `NULL`:

   $\boxed{\textbf{UQ} \leftarrow (\forall er \in nr(cookies.name) \neq er(cookies.name) \wedge nr(cookies.host) = er(cookies.host) \wedge nr(cookies.path) = er(cookies.path)) \wedge nr(cookies.name) \neq NULL \wedge nr(cookies.host) \neq NULL \wedge nr(cookies.path) \neq NULL)}$

These predicates in conjunction with each other forces the tester to satisfy the test requirement and create test data that will be accepted by the database. This example shows that, for one test case, manually writing tests to cover a whole criterion can be tedious and automated test data generators will expedite the process. The following section review these automated techniques for database schemas using these coverage criteria.

2.3.6 Schema Test Data Generation

Automation techniques for schema-based testing and its integrity constraints was also proposed and created by McMinn et al. [7] using a framework called *SchemaAnalyst*. The framework generates test data to satisfy

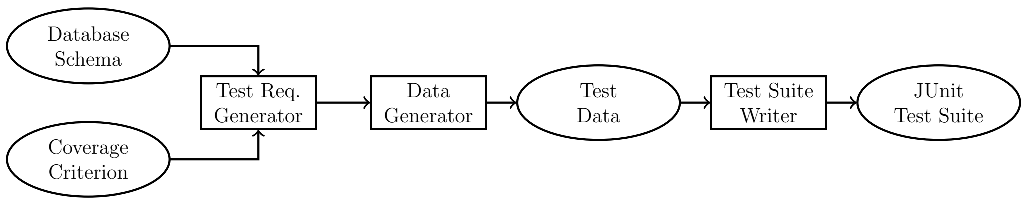


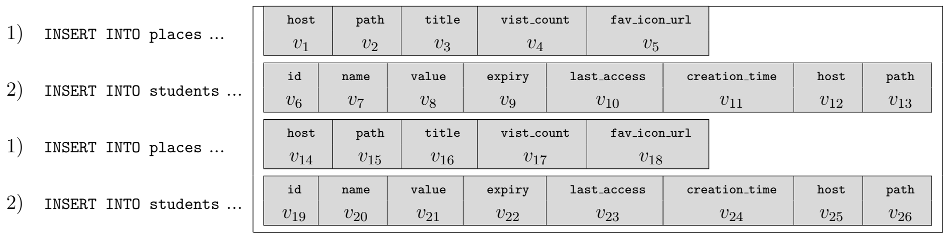
Figure 2.15: The inputs and outputs of the *SchemaAnalyst* tool

and cover any of the family of coverage criteria stated in the previous section to generate unit tests automatically. With a given schema and coverage criteria, the *SchemaAnalyst* will generate test requirements (i.e., the logical predicates stated in previous section) and generate the test data automatically, as illustrated in Figure 2.15. The framework implements two search methods for generating the test cases, a random technique and search-based technique. The framework generates test values for SQL `INSERT` statements to check if the database will accept or reject the `INSERT`s.

The Alternating Variable Method (AVM) is the search technique that was implemented into *SchemaAnalyst* to automatically generate test data [16, 7]. It works to optimise a vector of test values according to a fitness function. Figure 2.16b shows the arrangement of the values of the test case in part (a) into the vector $\vec{v} = (v_1, v_2, \ldots, v_n)$.

```
1)  INSERT INTO places(host, path, title, visit_count, fav_icon_url)                          ✓
    VALUES('aqrd', 'xj', 'vnobtpvl', 0, 'dmnofpe');
2)  INSERT INTO cookies (id, name, value, expiry, last_accessed, creation_time, host, path)   ✓
    VALUES (0, 'ddfvkxnjg', '', -332, -333, -1050, 'aqrd', 'xj');
1)  INSERT INTO places (host, path, title, visit_count, fav_icon_url)                         ✓
    VALUES ('te', '', '', -40, 'vfbtnwimd');
2)  INSERT INTO cookies (id, name, value, expiry, last_accessed, creation_time, host, path)   ✓
    VALUES (1, 'kavd', '', 0, NULL, 165, 'aqrd', 'xj');
```

(a) *SchemaAnalyst* generated test case example, which consists of `INSERT` statements for a database instantiated by the *BrowserCookies* schema in Figure 2.12. The ✓ denote the data contained within each `INSERT` statement satisfied the schema's integrity constraints and was accepted into the database.


1) INSERT INTO places ...
2) INSERT INTO students ...
1) INSERT INTO places ...
2) INSERT INTO students ...
host path title vist_count fav_icon_url
v_1 v_2 v_3 v_4 v_5
id name value expiry last_access creation_time host path
v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13}
host path title vist_count fav_icon_url
v_{14} v_{15} v_{16} v_{17} v_{18}
id name value expiry last_access creation_time host path
v_{19} v_{20} v_{21} v_{22} v_{23} v_{24} v_{25} v_{26}

(b) The vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ representation used by random and fitness-guided search techniques for finding the test data for each `INSERT` forming the test in part (b).

Figure 2.16: A Test Case For the Students Schema

The AVM, illustrated in Figure 2.17, starts by initialising each value in the vector at random. Next, it proceeds through the vector, sequentially making a series of "moves" (adjustments) to each variable value. These are either small exploratory or large jumps in one direction in a pattern moves, exploring the fitness landscape for improvements. It performs moves until a complete cycle through the vector during which no move successfully yielded a fitness improvement. At this point the algorithm may restart with a new randomly initialised vector. The AVM terminates when either the required test vector has been found, or a pre-defined resource limit has been exhausted (e.g., some number of fitness function evaluations).

```
while ¬ termination_criterion do
    RANDOMIZE($\vec{v}$)
    $i \leftarrow 1; c \leftarrow 0$
    while $c < n \wedge \neg termination\_criterion$ do
        $\vec{v}' \leftarrow$ MAKEMOVES($v_i$)
        if FITNESS($\vec{v}, r'$) < FITNESS($\vec{v}, r$) then
            $\vec{v} \leftarrow \vec{v}'$; $c \leftarrow 0$
        else
            $c \leftarrow c + 1$
        end
        $i \leftarrow (i \bmod n) + 1$
    end
end
```

Figure 2.17: the AVM algorithm that automatically generate, according to some coverage criterion r, a vector $\vec{v}$ of variables appearing in the `INSERT` statements of a test case for database schema integrity constraints.

Traditionally, the AVM has been applied to numerical test data generation [52]. However, databases can have many data types, including strings and dates. These are handled by representing the variable as a "sub-vector" of the overall main vector. That is, the variable itself is broken into a series of variables, each optimised by the AVM. For instance, a string is represented as a variable-length sequences of characters. Furthermore, values in databases may also be "`NULL`". The AVM adaptation for database schemas therefore includes a "`NULL`-move", whereby the value is shifted to `NULL` and the effect is checked on the fitness function. This move is reversed if fitness does not improve. Thus, the AVM is equipped to generate test data for schemas including handling for variable-length strings, dates, and `NULL` values.

The fitness function within the AVM uses the coverage requirement of a given coverage criteria (e.g., the satisfaction or violation of a particular integrity constraint). The fitness function for each coverage requirement is constructed using *distance functions* similar to those employed in traditional test data generation for programs [43, 100]. For example, for satisfaction of the `CHECK` constraint "`expiry > last_access`" for `INSERT` statement 2 of Figure 2.16a, the distance function $v_{10} - v_9 + K$ is applied and minimised (where K is a small positive constant value, and v_{10} is the vector value of Figure 2.16b). Conversely, for violation of the constraint, the distance function $v_9 - v_{10} + K$ is used. `NOT NULL` constraints are easily solved using the AVM via the aforementioned `NULL`-move. The fitness function assigns a high (i.e., poor) fitness when a `NULL`/non-`NULL` value in the vector that is contrary to that required. Primary key, `UNIQUE`, and foreign key constraints involve ensuring that certain values are the same or different to those appearing in prior `INSERT` statements of the test, depending on whether the constraint is to be satisfied or violated. For instance, suppose in the test of Figure 2.16a, the fourth `INSERT` statement was required to satisfy the primary key of the `cookies` table, by having a different `id` column value. In this case, the distinct values are computed with the distance such that $|v_{19} - v_6| + K$ would be applied.

Furthermore, the AVM introduced and used by McMinn et al. [7], initialises the vector to a series of default values chosen for each type (e.g., zero for integers and empty strings for `VARCHAR`) and only randomising the vector on the method's restart, referred as AVM-D in the following chapters. That is because the AVM can get stuck in local optima when initialised with random values trying to match values for `UNIQUE`s, `PRIMARY KEY`s, and `FOREIGN KEY`s. The use of default values increases the likelihood of inducing matching column values from the outset, speeding the test data generation. For example, Figure 2.18 shows a *SchemaAnalyst* produced JUnit test case generated by AVM with default values and the test case has equal test requirement as the test case in Figure 2.16a.

```
@Test
public void test8() throws SQLException {
// 11-cookies: UNIQUE[name, host, path] for cookies - all cols equal except name - Clause-AICC

// prepare the database state
assertEquals(1, statement.executeUpdate(
"INSERT INTO \"places\"(" +
"    \"host\", \"path\", \"title\", \"visit\_count\", \"fav\_icon\_url\"" +
") VALUES (" +
"    ’’, ’’, ’’, 0, ’’" +
");"));
assertEquals(1, statement.executeUpdate(
"INSERT INTO \"cookies\"(" +
"    \"id\", \"name\", \"value\", \"expiry\", \"last\_accessed\", \"creation\_time\", \"host\", \"path\"" +
") VALUES (" +
"    0, ’’, ’’, 0, 0, 0, ’’, ’’" +
");"));

// execute INSERT statements for the test case
assertEquals(1, statement.executeUpdate(
"INSERT INTO \"places\"(" +
"    \"host\", \"path\", \"title\", \"visit\_count\", \"fav\_icon\_url\"" +
") VALUES (" +
"    ’a’, ’’, ’’, 0, ’’" +
");"));
assertEquals(1, statement.executeUpdate(
"INSERT INTO \"cookies\"(" +
"    \"id\", \"name\", \"value\", \"expiry\", \"last\_accessed\", \"creation\_time\", \"host\", \"path\"" +
") VALUES (" +
"    1, ’a’, ’’, 0, 0, 0, ’’, ’’" +
");"));
}
```

Figure 2.18: An Example JUnit generated test by *SchemaAnalyst*. This test satisfies the all constraints and exercising the `name` column of the composite `UNIQUE` in the `places` table of the *BrowserCookies* schema. This test was generated using AVM method with default values.

The random search for relational schema testing simply involves repeatedly generating vectors with random values until the required vector is found, or other resources limit was exhausted. The random technique used by McMinn et al. [7] called Random$^+$ was not so naive because technique utilised a pool of constants mined from the schema (i.e., values within `CHECK` constraints). Therefore, when a random value is required, a value may be selected from this pool or generated freely at random, depending on some probability. The purpose of the pool is to help each algorithm satisfy and violate `CHECK` constraints in the schema for some requirement of a coverage criterion. The pool of constants was also utilised by the AVM to generate test data for database schemas.

McMinn et al. [7] empirically evaluated AVM against a random search technique. The experiment was conducted on all the above criteria and on three different DBMSs, HyperSQL, PostgreSQL, and SQLite. Their results showed there was no significant difference regarding coverage between the databases for each test data generator. The only difference that was observed, even though there is no significance, was the test suites differ for SQLite compared to other DBMSs. That was because SQLite varies in implementing `PRIMARY KEY`s that accepts `NULL` as a value.

The AVM was significantly better than the random search in experiments conducted on a wide range of schemas, including those with complex integrity constraints and many tables [16, 7]. Furthermore, AVM results attained 100% coverage for different criteria for most schemas studied. However, random search never achieved full coverage for any schema, obtaining less than 70% in some instances.

They evaluated the fault-finding capabilities of each criterion using mutation analysis for database schemas. The mutation analysis simply adds, removes, and exchanges integrity constraints within a schema. Their results only included effective mutants, which will be discussed in detail in the following section. The results showed that the higher the test suite's coverage criterion in the subsumption hierarchy, the more faults are detected. That is especially for the AVM generated test suites. They observed that column coverage criteria were better at detecting mutants that introduce omission type faults. Moreover, constraint coverage criteria were better at detecting mutants that introduce commission type faults. Therefore, different criteria complement each other. This observation allowed them to empirically evaluate combining different coverage criteria regarding fault detection. Hence, their results show that combining criteria at the top of each hierarchy resulted in the best fault-finding capability with 94% killed mutants. However, this result was second using AVM and default values that resulted with the combination of ClauseAICC, UCC, and ANCC. Because they found that default values that were used in AVM were influencing the detection rate compared to random values generated by Random$^+$. Thus, the use of default values resulted in test cases sharing many similar values, thereby lowering diversity and hindering the fault-finding capability of the tests.

The *SchemaAnalyst* framework required some future work as reported by its authors. That is, incorporating new algorithms that improve the effectiveness and efficiency compared to the current techniques. Furthermore, improving the generated test suites regarding fault-finding capabilities and reduction.

This thesis will use *SchemaAnalyst* to automatically generate test data and evaluate many techniques in the following chapters. Both the AVM and Random$^+$ will be used to as a baseline for evaluating other generators in the following chapters. The procedures and recommendations of McMinn et al. [7] will be followed closely. However, the thesis research differs from their work as it will evaluate multiple test data generators effectiveness rather than evaluating test coverage criteria.

2.3.7 Schema Mutation Analysis

Mutation analysis can help evaluate and estimate the fault-finding capabilities of a test suite. That is, systematically seeding the database schema with faults using mutation operators and running the test suite against mutated schema. Mutation analysis for database schema was first introduced by Kapfhammer et al. [16] to validate the quality of test cases that exercise the database integrity constraints. The operators add, remove, or exchange the main integrity constraints on columns. However, this was extended by Wright et al. [101] with operators that exchanging the columns and relational operators in `CHECK` constraints, and mutating `FOREIGN KEY`s. Table 2.1 show the integrity constraints mutation operators by their creator.

By	Operator Name	Description
Kapfhammer et al.	PKColumnA	Adds a `PRIMARY KEY` to constraint to a column
	PKColumnR	Removes a `PRIMARY KEY` constraint from a column
	PKColumnE	Exchanges a `PRIMARY KEY` constraint with another column
	FKColumnPairR	Removes a column pair from a `FOREIGN KEY`
	NNA	Adds a `NOT NULL` constraint to a column
	NNR	Removes a `NOT NULL` constraint from a column
	UColumnA	Adds a `UNIQUE` constraint to a column
	UColumnR	Removes a `UNIQUE` constraint
	UColumnE	Exchanges a `UNIQUE` constraint column with another
	CR	Removes a `CHECK` constraint
Wright et al.	FKColumnPairA	Adds a `FOREIGN KEY` to a pair column
	FKColumnPairE	Exchanges a `FOREIGN KEY` from one column pair
	CInListElementR	Removes an element from an IN `CHECK` constraint
	CRelOpE	Exchanges a relational operator in `CHECK` cosntraint

Table 2.1: Kapfhammer et al. [16] and Wright et al. [101] mutant operators.

`PRIMARY KEY` mutants manifest in three ways: add column, exchange column, or remove column. That is, a schema can be mutated by exploring each column and applying these operators. If the column has a `PRIMARY KEY`, it can be removed to produce a mutant, or the `PRIMARY KEY`'s column is exchanged with another column. If no `PRIMARY KEY` constraint constraint is present in the table then it will be added to column. Therefore, having the PKColumnA operator for addition, PKColumnR operator for removal, and PKColumnE operator for exchanging. These operators are illustrated in the first row of Table 2.2 were the `PRIMARY KEY` is in the `places` table of the *BrowserCookies* schema.

Original Constraints	Add Mutation	Remove Mutation	Exchange Mutation
`PRIMARY KEY (host, path)`	`PRIMARY KEY (host, path, title)`	`PRIMARY KEY (host)`	`PRIMARY KEY (host, visit_count)`
`UNIQUE (name, host, path)`	`UNIQUE (name, host, path, expiry)`	`UNIQUE (host, path)`	`UNIQUE (name, host, value)`
`host TEXT NOT NULL`	-	`host TEXT`	-
`title TEXT`	`title TEXT NOT NULL`	-	-
`FOREIGN KEY (host, path) REFERENCE places(host, path)`	`FOREIGN KEY (host, path, value) REFERENCE places(host, path, title)`	`FOREIGN KEY (host) REFERENCE places(host)`	`FOREIGN KEY (host, name) REFERENCE places(host, title)`
`CHECK (last_access >= creation_time)`	-	removed	`CHECK (last_access < creation_time)`
`CHECK (gender IN ('Male', 'Female', 'Uknown'))`	-	`CHECK (gender IN ('Male', 'Female'))`	-

Table 2.2: Integrity constraints from the *BrowserCookies* schema in Figure 2.12 with the applying mutation operators in Table 2.1

`UNIQUE` mutants manifest in the same way as `PRIMARY KEY` mutants. That is, adding, exchanging, or removing columns with `UNIQUE` constraints, as illustrated in the second row of Table 2.2. `NOT NULL` mutants can manifest in two ways, either by removing a `NOT NULL` if declared on the column or adding a `NOT NULL` constraint if column does not have its declaration (row 3 and 4 of Table 2.2).

`FOREIGN KEY` mutants manifest in three ways, similar to a `PRIMARY KEY` and `UNIQUE` constraints. However, `FOREIGN KEY` addition or exchange must have matching data types of the paired columns. For example, the fifth row of Table 2.2 show adding `value` and `title` columns in the `FOREIGN KEY` can mutate the schema and both columns are `TEXT` data type.

`CHECK` mutants differ from other constraints and can manifest in many ways. That is because `CHECK`s can use relational operators (e.g., $=, <, >, \leq, \geq$) to compare two columns, or the `IN` operator which is equivalent to multiple `OR` operators. The `IN` operator checks the value against a set of values, if they do not match, then the value is rejected. Therefore, `CHECK` mutants manifest with the following: (1) removing the `CHECK` constraint from the schema using the CR operator (e.g., row 6 of Table 2.2); (2) exchanging relational operators within the predicate, if applicable, using the CRelOpE operator (e.g., row 6 of Table 2.2); (3) removing a value from the `IN`, if applicable, using the CInListElementR operator (e.g., row 7 of Table 2.2).

The aforementioned mutation operators were implemented into *SchemaAnalyst* to measure the effectiveness of fault-finding of generated test suites. Thus, making this framework the state-of-the-art and includes both test data generation and mutation analysis techniques. These operators were used by McMinn et al. [7] for evaluating the different coverage criteria and test data generator, the AVM and Random^{+}, stated in previous section. However, their results did not include all mutants, and they removed ineffective mutants that are: *equivalent*, *redundant*, and *quasi-mutants* (referred to as "still-born"). An equivalent schema mutant has equal functionality to the original schema. Redundant mutants are the same as other mutants in regard of functionality. Quasi-mutants are schemas that are invalid or infeasible, depending on the DBMS implementation. For example, if a DBMS implementation forces a `FOREIGN KEY` to reference to only `UNIQUE`s or `PRIMARY KEY`s (e.g., PostgreSQL), then any mutant that violates this implementation is a quasi-mutant [101]. Therefore, *SchemaAnalyst* implements the detection of ineffective mutants and it will be used in the following chapters' experiments.

2.4 Limitations and Research Gaps

This section reviews the literature to identify research gaps in automated test data generation, particularly in schema testing. It starts with reviewing inefficiencies of search-based test generators. It also reviews the challenges and improvements in test comprehension. It then reviews some traditional methods for test suite reduction and their potential uses and limitations in decreasing the test suite sizes for relational database schema testing.

2.4.1 Search-Based Test Data Generation Inefficiencies

Shamshiri et al. [102, 103] empirically studied two evolutionary algorithms compared to two random search techniques in object-oriented classes. The first technique is a standard genetic algorithm (GA), the other is based on a chemical reaction optimisation (CRO). The two random techniques are naive random and Random$^+$ (i.e., a random search that utilises mined values from the class). Their results showed that the GA and CRO are comparable regarding the coverage in some cases, and the rest CRO significantly covered more branches. The results also showed that the random techniques covered less branches. However, and surprisingly, their analysis on both the search-based techniques generated tests with less diverse data compared to the random search techniques. Because the search-based techniques spend a large amount of time evaluating the neighbourhood of existing solutions (i.e., test data), however both random searches keep moving in many neighbourhoods creating new diverse test data. Random search was more efficient in generating new tests, but the search-based techniques generated tests covering more complex branches. Furthermore, Random$^+$ showed to generate better tests compared to both search-based methods with plateau branches (i.e., a branch with non-gradient distance landscape) which are the majority in their subjects. Therefore, they suggested that random search can be optimised and utilised in generating tests with higher coverage.

Shamshiri et al. [102] results and conclusions were similar to Sharma et al. [104] that they compare random testing to a systematic technique specific for container classes called Shape Abstraction. The test context was on container classes that are list or set representation classes. Their study and results show that random testing preform efficiently and create long sequences compared to the Shape Abstraction. Therefore, random testing needed fewer computation resources compared to the more specific technique.

In summary, the use of random test data generation can be more beneficial than search-based techniques in regard of test data diversity. Random methods can be engineered to be more specific for the domain context to gain comparable coverage to other state-of-the-art techniques, and improving fault-finding capabilities, such as the repair methods that match and mismatch values of `INSERT`s in EvoMaster [95]. Moreover, according to McMinn et al. [7] the use of default values with AVM generated tests that are efficient. However, these values resulted in weakening the fault-finding capabilities of the strong combined coverage criteria. Thus, random values can be used to generate more diverse and effective tests. This might affect the human test oracle with many tests that take long time to run and evaluate. Therefore, in the following, chapters different values will be investigated and explored for effectiveness and overall human oracle cost in the context of database schema testing.

2.4.2 Test Comprehension

Automated test data generators can help testers to avoid the tedious and error-prone task of manually writing tests for a database schema. Prior work has shown that automatically generated tests can effectively cover the schema and detect synthetic schema faults [7, 101, 105]. Yet, testers must still act as an "oracle" for a test when they judge whether it passed or failed [24], a challenging task that is often overlooked. Especially, comprehending each test case outcomes and whether the observed behaviour is correct or otherwise.

Test comprehension is a frequently studied issue. For instance, Li et al. surveyed 212 developers and more than half reported difficulty with understanding unit tests [106]. Interestingly, the survey reported that only ∼53%

of developers do ‘fairly often’ or ‘always’ write tests, and ∼44% of developers reported that they ‘never’ or ‘rarely’ change tests. Inferring, that tests are always created, but they are difficult to understand, making changing them difficult. The paper also proposed an automatic test documentation tool called ‘UnitTestScribe’ that utilises static analysis, natural language processing, backward slicing, and code summarisation techniques to automatically generate comments/documentation for test cases. They again surveyed developers to evaluate their tool and the results showed their technique was able to generate easy to read informative documentation for tests with minimum redundancy. Thus, helping developers to understand tests.

Li et al. [107] used a tagging technique called stereotypes for each test case within a test suite, which they called TeStereo. A stereotype is a comment or a tag that reflects the role of a program (i.e., a class or method). Therefore, their technique can tag a test case with description. That is, a test case that tests a boolean can be tagged with a “Boolean verifier” and a description of “Verifies boolean conditions”. To empirically evaluate their technique with a human study (71 participants) with a group with no tagging and the other group with tagging. Their results showed that ∼58% agreed that TeStereo was complete with no missing information, ∼68% agreed that TeStereo was concise with no redundant information, and ∼56% agreed that TeStereo was easy to read and thus expressive. Of the 71 participants, 25 are active Apache developers that responded with feedback that TeStereo reports are useful for test case comprehension tasks. However, this was not for testing database schemas and there is a gap in the literature in that context.

Similar to UnitTestScribe, Linares-Vásquez et al. [108] created a documentation tool for database applications called ‘DBScribe’. This tool statically analyses the application code and the database schema to infer the usage of queries to generate comments. The comments are automatically generated and added to the application code to help developers understand the interactions between the embedded query and the database. For example, it would add the following comment: “It inserts the <attri> attributes into table <table>” as a template for an `INSERT` statement. They surveyed their tool with 52 participants, a mix of students and professional developers.

Their survey results showed that 'DBScribe' had the following participants agreements: ∼65% agreed that it generate complete information. ∼70% agreed that generated comment was concise without redundant information. ∼77% agreed that the comments were easy to read, and ∼92% agreed that it would be useful. Therefore, automatically commenting database interactions are useful and important to help with debugging and maintenance. However, there are no indication of this that will help with testing and understanding the test data, especially for automated test data generation.

Cornelissen et al. [109] studied visualisation techniques to help with understanding tests. However, they have not evaluated their technique with a human study. Furthermore, Smith et al. [110] applied a multi-plot to show the test suite order for the purpose of test prioritisation. This was to help testers to evaluate and compare the effectiveness of test suite order. They conducted an informal study with a senior researcher and two postgraduate students. They inferred that this technique may help testers with plots rather than raw data.

All the above work does not concern with test data and how it impacts the comprehension of tests. This thesis focuses on understanding automatically generated test data rather than the visualisation or tagging of tests. Therefore, the following explore related research in the test readability to increase test comprehension.

Test readability motivated Afshan et al. [69] to incorporate it into test data generation. They applied a natural language model (LM) as an objective of the search-based technique (i.e., part of the AVM fitness) to generate more readable string data. The LM works by assigning a probability score to a string depending upon its likelihood of occurring as part of a language, by checking how well a string is formed. The process of LMs starts by loading a corpus (documents) before the evaluation process to train the language model. LMs are often used in Natural Language and Speech Processing research. Therefore, Ashfan et al. [69] used the character-based language model to estimate the probability of each character based on the character immediately precedes it. They implemented their technique to generate test data for Java programs using the IGUANA framework and evaluated the readable test data on a human study. They compared the AVM-LM with randomly generated test data. Their results showed that the language model generated significantly more readable test data regarding correct answers (i.e., accuracy of judgements) for only three case studies. However, the rest of the cases (i.e., 14 case studies) did not show any hindrance to accuracy. They also evaluated the test evaluation duration, and the results show that participants responded faster when presented with LM data compared to random data.

Daka et al. [24] improved automatically generated test cases by increasing readability using predictive models and incorporated this into the EvoSuite framework. They built their predictive model based on results of crowd source participants generating 15,669 human readable scores using 450 automated and manual tests. Then they used this model to generate readable variables and values to replace the automatically generated text in tests. In their study, they compared generated default tests with the more readable optimised tests in regard of readability scores and a human study to evaluate readability. Their results indicated that the predictive model was 2% more readable on average, 69% of human participants preferred the readable optimised tests, and participants answered questions 14% faster with no change in accuracy.

Rojas et al. [111] conducted two human studies with students and professionals to evaluate if automated tests are helpful to developers compared to manually written tests. Their first human study with 41 students indicated that EvoSuite, the automated testing tool, supports developers with tests that have more than 14% coverage and 36% less time spent compared to manual testing, with branch testing coverage criteria. Their second study was a think-aloud study, asking participants to describe their thought processes aloud to obtain inferences, reasons, and decisions made by participants. This study included five professionals and confirmed that automated tools support testers. However, the generated tests were hard to understand and difficult to maintain. Therefore, both studies showed that automatically generated tests need to be more usable (i.e., readable and understandable).

In another research, Daka et al. [71] investigated variable naming and how it affects the readability and understandability of test cases. In their study, they motivated their work as automatically generated test case that are name d"test0". Thus, they proposed a technique that uses the summarises the coverage goals as the name of the test, which was also integrated into the EvoSuite framework. For example, their technique will change "test0" to either "getTotalReturningPositive" or "getTotalReturningNegative" depending on the coverage goal. They evaluated this technique against manually written test names with 47 participants. The study results showed that the participants agreed similarly and disagreed less with synthesised names which are equally descriptive to manually written tests names. Participants are also tasked with matching the code to the tests. This showed that the participants are slightly more accurate and faster with synthesised names compared to manually written names. Moreover, participants were more accurate at identifying the relevant tests for a given code using synthesised test names compared manually written test names.

Grano et al. [112] explored and studied the readability of manually written test cases compared to the code under test and automatically generated test cases. They used a readability model to compute the readability of tests and the code under test. This model was created by Scalabrino et al. [113] to evaluate program code readability. Grano et al. study showed that manually written test cases were significantly less readable than the code under test. However, manually written tests are significantly more readable than automatically generated tests with small effect size. Therefore, their conclusion was that developers tend to write less readable test cases and automated tools generally produce the worst readable test cases. This work might not be applicable with database schemas due to SQL`CREATE TABLE` commands have nearly equal structure while `INSERT`s values are different in readability, making unfair comparison.

These studies examined test comprehension in the context of traditional programs. In the context of understanding the SQL language, some researchers studied human errors in database query languages [114, 115, 116, 117]. However, there is no work on test comprehension in the domain of database schemas and SQL statements. This is surprising, since there are many prior methods for automatically testing and debugging a database. Importantly, previous work did not characterise the impact of different test inputs on a human oracle, especially in the context of automated testing and more particularly in database schema testing. Therefore, trying to evaluate the generated test data and its human understanding affects while determining the test's behaviour is limited in the research of software testing. Consequently, it is important to identify and infer the characteristics that make test cases easy to understand for testers.

This thesis intends to identify the characteristics of understandable schema test cases. Thus, the thesis will present techniques with test readability incorporated for database schema testing (e.g., language model and readable values). This will help identify comprehension factors of different readable test inputs. However, as this thesis domain is database schema testing, other test readability methods such as readable variable names [71] are not applicable with `INSERT` statements. Rojas et al. [111] used the think-aloud protocol that helped to identify the difficultly of understanding automated program tests. Therefore, this thesis will utilise the think-aloud protocol to go further and identify factors of understandable SQL tests.

2.4.3 Test Suite Size

Test suites can contain many test cases that take a long time to run and require longer times for testers to evaluate the whole test suite. The test cases can have overlaps of requirements based on their test requirement. For example, one test case can subsume and cover one or more test cases (i.e., covering their test requirements). Thus, a test suite can be reduced using reduction techniques to have representative test suite. The reduced test suites will also help decrease the quantitative human oracle costs.

Reducing a test suite is equivalent to the minimal set cover problem, which is NP-complete [118]. There are several heuristics capable of effectively reducing test suite size to support developers. Yoo and Harman [119] surveyed prior work on ways to reduce suites by selecting a representative subset of test cases. These included Random, Greedy [120], HGS [25], Greedy

	r_1	r_2	r_3	r_4	r_5	r_6
t_1	X	X	X			
t_2	X			X		
t_3		X			X	
t_4			X			X
t_5					X	
	T_1	T_2	T_3	T_4	T_5	T_6

Figure 2.19: Example of test cases $\{t_1, \ldots, t_5\}$ and test requirements $\{r_1, \ldots, r_6\}$, an input for test suite reduction methods (excerpted from [122]).

Essential (GE), and Greedy Redundant Essential (GRE) [121]. This section will review these reduction techniques which will be used in the following chapters to evaluate against a new reduction technique. Also, Figure 2.19 will be used to illustrate how the reduction techniques work. This example figure shows a test suite with five test cases $\{t_1 \ldots t_5\}$ and five test requirements $\{r_1 \ldots r_6\}$, where the test cases have different, yet overlapping, coverage of the test requirements.

A random test suite reduction is a simple heuristic that is often effective at reducing test suites [119]. As illustrated in Figure 2.20a, this reduction method starts with an empty test suite, adding test cases from the original test suite so long as they cover new test requirements, and continuing until all test requirements are covered. A Greedy heuristic, in some literature is called additional greedy and illustrated in Figure 2.20b, works in a similar loop to produce a smaller test suite, but instead of selecting test cases at random from the original test suite, it selects the next previously unconsidered test case that covers the most uncovered test requirements (the *max_cov*() method in the algorithm) [119]. In the example from Figure 2.19, Greedy selects t_1 first. Since the remaining test cases all cover one remaining requirement, this reduction method will select them at random until all requirements are covered, yielding a reduced test suite of four test cases.

Another well-known approach, called HGS, was developed by Harrold, Gupta, and Soffa [25], and illustrated in Figure 2.20c. It works by creating test suites containing test cases that cover each test requirement, i.e. $T_1 =$

```
1: while $R = \phi$
2:     $t \leftarrow$ randomly selected test case
4:     $RTS = RTS \cup t$
4:     $T = T - \{t\}$
4:     $R = R - \{r|(t,r) \in S\}$
4: return $RTS$
```

(a) Random

```
1: while $R = \phi$
2:     $t \leftarrow$ test case with max_cov($t$)
3:     $RTS = RTS \cup t$
4:     $T = T - \{t\}$
5:     $R = R - \{r|(t,r) \in S\}$
6: return $RTS$
```

(b) greedy

```
 1: $T_1, T_2 ... T_n$ associated test sets for $r_1, r_2 ... r_n$
 2: $curret\_car \leftarrow 1$
 3: $RTS = RTS \cup T_i, car(T_i) = curret\_car$
 4: $mark(T_i \cap RTS) = true$
 5: while $curret\_car \leq max(cardinality)$
 6:     $curret\_car \leftarrow curret\_car + 1$
 7:     while $car(T_i) = curret\_car \wedge \neg mark(T_i)$
 8:         $list \leftarrow t_i \in T_i$
 9:     $RTS = RTS \cup select(curret\_car, list)$
10:     $mark(T_i \cap RTS) = true$
11: return $RTS$
```

(c) HGS

Figure 2.20: Test Suite Reduction algorithms. T denote the set of test cases. R denote the set of test requirements. S indicate the relation between a test case t that satisfies a requirement r, $S = \{(t, r)|t$ satisfies $r, t \in T$, and $r \in R\}$. RTS is the reduced test suite.

$\{t_1, t_2\}$, covering r_1; $T_2 = \{t_1, t_3\}$ covering r_2, up to $T_6 = \{t_4\}$, covering r_6. HGS starts by adding test cases to the reduced test suite from the test suites $T_1 \dots T_n$ with cardinality 1 (Line 3 of the algorithm). In the example, test suites with cardinality 1 are T_4 and T_6, involving test cases t_2 and t_4, which result in the coverage of $\{r_1, r_4\}$ and $\{r_3, r_6\}$, respectively. HGS then "marks" test suites that also cover these requirements (i.e., T_1 and T_3) so they are not considered by further steps of the algorithm (illustrated as the $mark()$ method in Line 4). HGS then repeatedly selects the test cases in unmarked test suites of increasing cardinality. In the example, unmarked test suites of cardinality 2 are T_2 and T_5, with t_3 the only test case to occur in both and evaluated using the $select()$ method, and thus added to the reduced test suite. Since t_3 covers r_2 and r_5, all test requirements are now covered, and the algorithm terminates with the reduced test suite containing three tests — one fewer than Greedy. HGS avoids selecting t_1, which is challenging for Greedy, thus leading to Greedy being less successful than HGS at reducing this example test suite. Both HGS and Greedy were extended by other researcher for more complex examples. Moreover, experimental studies showed that HGS and Greedy significantly reduced the size of test suites [123, 121].

Both GE and GRE are variants of the Greedy algorithm that was developed by Chen and Lau [121]. The GE stands for "Greedy Essential", which starts by selecting essential, or "irreplaceable", test cases first then applies the standard greedy algorithm. For example and referring back to Figure 2.19, the essential tests are t_2 and t_4 because r_4 and r_6 are only satisfied by these test cases and will be selected first. Then the greedy will consider t_1, t_3, and t_5, however, with the unsatisfied test requirements, t_3 will be selected as it has the highest covered test requirements. Conversely, GRE (i.e., Greedy Redundant Essential) removes any redundant tests that the essential tests already covered (e.g., removing t_1 to consider other test cases), the Greedy will only consider t_3 and t_5 selecting only t_3. In this example, GRE and GE are equal to HGS because the overlap was small. However, in their study, they showed that GE and GRE selected smaller or equal set of tests compared to HGS depending on the overlap size (i.e., overlap of requirements). However, HGS was able to reduce the test suite more than both techniques with different overlaps. Their technique requires knowing the overlap prior to reducing the test suite. Therefore, with test data generation and randomisation, overlaps are difficult to predict and falling to the original technique is the safest option. The study suggested that no technique is better than the other as they are approximations rather than precise algorithms.

Tallam and Gupta [124] developed a greedy algorithm called *delayed greedy*. It was based on Formal Concept Analysis (i.e., deriving a concept hierarchy) of relations between tests and requirements. That is because a Greedy algorithm makes early selections of redundant test cases (e.g., selecting t_1). Delayed greedy first transforms the relations into a hierarchy that removes tests that are a subset of another test case based on requirements. Second, it removes the requirements that are also a subset of other test requirements. Then, the greedy algorithm is applied on transformed set. For instance, if test case t_i has a superset cover t_j test requirements then t_j will be removed. After that, if r_i test cases are a subset of r_j test cases then r_i will be removed. Lastly, the greedy algorithm will run on the rest of test cases and requirements. Their empirical evaluation showed that their technique had smaller or equal minimised test suites compared to Greedy and HGS.

Many others have used these methods as building blocks for new reducers (e.g., [125, 126, 127]). Some applied integer linear programming (e.g., [128, 129]) or evolutionary algorithms (e.g., [130]) to the problem of test suite reduction. Finally, Yoo and Harman used multi-objective search with test reduction, test coverage, and past fault-detection history as goals [131]. Vahabzadeh et al. [132] proposed a technique to minimize test cases with identifying equal test states to combine the assertions within tests into one test, reducing the whole test suite considerably and improving regression testing efficiency. All of this work has been applied to program code rather than database schema testing context. Therefore, there is a research gap of applying test suite reduction techniques into database schema testing.

In the context of database testing, there are several studies of database schema evolution (e.g., [35, 22, 133]), thereby motivating the need for efficient regression testing methods. Kapfhammer reduced test suites for database application tests using a greedy algorithm [134]. Similarly, Tuya et al. used a greedy algorithm to reduce the amount of data within databases for testing SQL `SELECT` queries [98, 135]. Haftmann et al. used a slicing technique to prioritise tests and reduce the number of database resets to improve the efficiency of regression testing [94]. However, unlike these examples of prior work, there are no work on database schema testing.

Finding tests that are understandable can decrease the oracle cost. However, the size of test cases and test suites can be large and will require longer times to evaluate. Many factors can influence the test suite size such as the combination of coverage criteria recommend by McMinn et al. [7] which was the strongest in regard of fault-finding capabilities. This will lead to many duplicate tests with different test requirements satisfying different coverage criteria (e.g., satisfying a `UNIQUE` column test and a `PRIMARY KEY` column test). Another factor is the duplication of `INSERT` statements, and they can be unnecessary to the test requirement. Therefore, removing such `INSERT`s will not affect the test case coverage. Thus, the following chapters will evaluate test suite reduction methods in database schema testing and identify improvements for such domain.

The benefits of reducing the test suite and the test cases is the execution speed. Especially with mutation analysis (i.e., mimicking regression testing and faults) because many mutants can be produced and requires the test suite to be executed each time. Another benefit is reducing the human oracle cost that can help promote the use of automated tools. However, there are no prior work in testing database schema and there are no prior work reviews test suite reduction with actual humans. Therefore, these are major research gaps that should be tackled and empirically evaluated.

2.5 Summary

Relational databases are the backbone of most software systems and it is considered the most valuable asset of any organisation. That motivates the importance of database design (i.e., schema) testing, which is essential to ensure the quality of a system. However, writing methodological tests manually using coverage criteria is tedious, error-prone, and time-consuming. Therefore, automating and generating tests can lower this cost [111]. However, testers must execute such tools and evaluate the produced tests. This requires the test data generation tools to produce effective test cases at high speeds that are understandable, and as short as possible while maintaining the coverage and fault-finding capabilities. That is, improving these tools and empirically evaluating them is crucial as they need to be applicable in real-world testing scenarios.

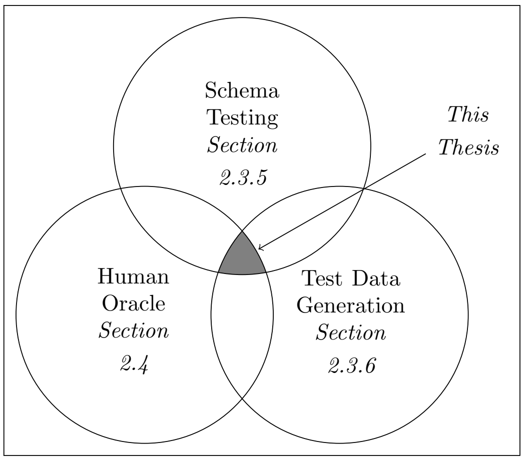

Schema Testing *Section 2.3.5*
This Thesis
Human Oracle *Section 2.4*
Test Data Generation *Section 2.3.6*

Figure 2.21: Identified gaps related to existing work in this literature review and the contributions of this thesis.

This literature review surveyed software testing concepts and the state-of-the-art test data generation techniques for both software programs and relational databases. It identified gaps in research that are crucial for database schema testing as illustrated in Figure 2.21 and listed as follows:

- Viable improvements for test data generation
- Lowering the human oracle cost with the following:
  - Improving the comprehension of generated test data
  - Reducing generated test suites

This thesis will empirically evaluate new methods that generate which are more efficient, effective, and less demanding of human oracle cost. First, evaluating and improving a domain specific technique with a hyper technique that utilises random and AVM. Secondly, identifying understandable factors of tests inputs with a human study and multiple variants of readable techniques in the context of database schemas. Finally, implementing an improved reduction technique for database schema testing and evaluated against traditional techniques, improving both regression testing and the human oracle cost.

Chapter 3

DOMINO: A Fast and Effective Test Data Generator

The content of this chapter is based on the published work during this PhD, and presented in the International Conference on Software Testing, Verification and Validation (ICST) 2018 [17].

3.1 Introduction

In the literature review chapter, the Alternating Variable Method (AVM) [15] is the state-of-the-art method for generating schema tests. The AVM is a search-based technique that receives guidance from a fitness function [16, 7]. However, the generation of schema tests with this search can be slow, particularly when it must locate columns that need to have identical values and then adjust those values until they are the same. To aid the process, prior work configured the AVM that can start with a series of "default" values, thus ensuring that matches are likely from the outset. Yet, this can introduce a lot of similarity across the different tests in the suite, hindering both its diversity and potential fault-finding capability.

Therefore, this chapter introduce a new test data generation technique for testing relational database integrity constraints, called DOMINO (DOMain-specific approach to INtegrity cOnstraint test data generation). The DOMINO technique utilises random search with a tailored approach that uses domain specific operators to generate test data. That is, the technique try to "fix" the randomly generated values for `INSERT` statements to satisfy the given coverage criteria. Thus, DOMINO leverage the knowledge of the schema and a coverage requirement to explicitly sets data values to be the same, different, or `NULL`, only falling back on random method when it must satisfy more arbitrary constraints.

The intuition is that random values can generate more diverse test data and increase the fault-finding capability of the generated test suite (i.e., a more effective tests). Also, the fixing of randomly generated can increase the efficiency of the generator. However, these claims need to be empirically evaluated. Therefore, this chapters empirically evaluates and analyse the technique, enabling a new variant of hybrid technique of DOMINO to be created, called DOMINO-AVM.

This chapter experimentally compare DOMINO to both AVM and a hybrid DOMINO-AVM method, using 34 relational database schemas hosted by DOMINO different DBMSs (i.e., HyperSQL, PostgreSQL, and SQLite). The results show that DOMINO generates data faster than both the state-of-the-art AVM and the hybrid method, while also producing tests that normally detect more faults than those created by the AVM.

The DOMINO method was developed in 2014 while implementing and designing the *SchemaAnalyst* framework by Professor Phil McMinn. This chapter evaluated and improved this technique during the PhD time. Therefore, the outlines and contributions of this chapter as follows:

1. Experiments showing that DOMINO is both efficient (i.e., it is faster than the AVM at obtaining equivalent levels of coverage) and effective (i.e., it kills more mutants than the AVM), in Section 3.4.

2. An informal analysis of finding faults capabilities with different test data generators (Section 3.4).

3. The creation of a new hybrid technique called DOMINO-AVM (Section 3.5).

4. Experiments comparing both techniques and the results showed that the DOMINO-AVM is not superior to DOMINO (Section 3.5).

To support the replication of this chapter's experimental results and to facilitate the testing of relational database schemas, the procedures were implemented into scripts that can re-run experiments and analyse results. Please follow Appendix B.1 for replication instructions.

3.2 Motivation

Prior work has shown that the AVM can generate test data for relational schemas [7], it is subject to inefficiencies. First, the method waste time cycling through column values that are not involved in any of the schema's integrity constraints. That is changing each column's value to improvement the fitness score. Secondly, the AVM get stuck in local optima, requiring restarts as stated in the literature review. Because it tries to find test data (i.e., solutions) in a small segment of the search space and after many iterative changes with no improvements to the fitness. Finally, the AVM spend time making incremental changes to a particular column value to match another value in the test data vector, with the purpose of satisfying or violating a `PRIMARY KEY`, `UNIQUE`, or `FOREIGN KEY` constraint. For example, matching a `FOREIGN KEY` to a `PRIMARY KEY` (i.e., already been generated) requires the AVM to incrementally change a random value to match the generated value. Such as match a '906' and '-908' will enforce the AVM to make many steps, even with "pattern" moves, to match them both.

The last two issues can be mitigated by first initialising the vector to a series of default values chosen for each type (e.g., zero for integers and empty strings for `VARCHAR`), and only randomising the vector on the method's restart [7]. This increases the likelihood of inducing matching column values from the outset. Hereinafter, this variant of the AVM will be referred as "AVM-D", and the traditional randomly initialised vector version as "AVM-R".

3.3 The Domino Test Data Generator

Given the inefficiencies identified in the test data generator for integrity constraints, we developed an alternative, tailored approach to the problem that uses domain knowledge. This new approach, called "Domino" (DOMain-specific approach to INtegrity cOnstraint test data generation), can replicate values in the test data vector for different constraint types, depending on the coverage requirement. The Domino algorithm in Figure 3.1 begins by initialising the test data vector at random. Henceforth, this technique will be called DOM-RND as it use randomly generated values (i.e., the "RND" of DOM-RND) and to distinguish Domino (i.e., the "DOM" of DOM-RND) from new variations of these techniques presented in the next chapters. The main loop then works according to the following intuition: Where a value needs to be the same as one of a selection of values already in the vector, choose a value from that selection at random and copy it (through the copyMatches function); else randomly select a new value instead through the randomizeNonMatches function (where the "new" value is chosen from the constant pool, as described in Section 2.3.5, or is a freshly generated value). `NOT NULL` constraints and `CHECK` constraints are handled separately through the setOrRemoveNullsfunction and the solveCheckConstraintsfunction, respectively.

```
RANDOMIZE($\vec{v}$)
while ¬ termination criterion do
    copyMatches($\vec{v}, r$)
    randomizeNonMatches($\vec{v}, r$)
    setOrRemoveNulls($\vec{v}, r$)
    solveCheckConstraints($\vec{v}, r$)
end
```

Figure 3.1: The Domino (i.e., DOM-RND) algorithm that automatically generate, according to some coverage criterion r, a vector $\vec{v}$ of variables appearing in the `INSERT` statements of a test case for database schema integrity constraints.

```
CREATE TABLE products (
  product_no INTEGER PRIMARY KEY NOT NULL,
  name VARCHAR(100) NOT NULL,
  price NUMERIC NOT NULL,
  discounted_price NUMERIC NOT NULL,
  CHECK (price > 0),
  CHECK (discounted_price > 0),
  CHECK (price > discounted_price));
```

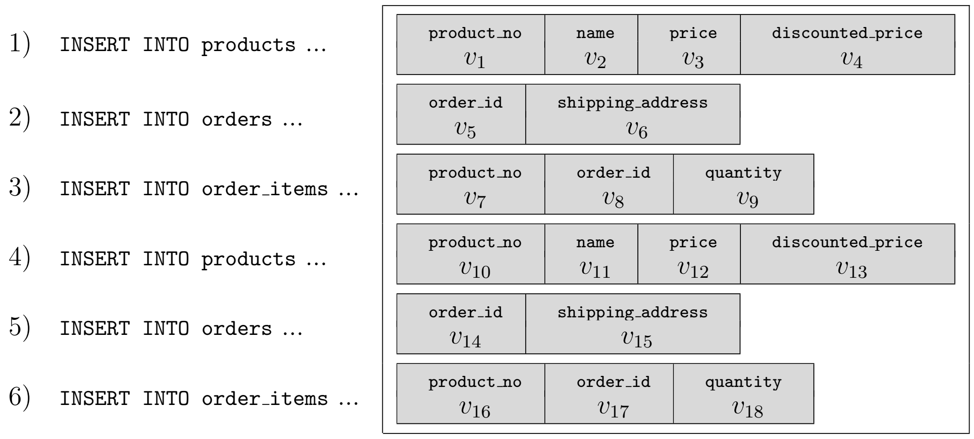
```
CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  shipping_address VARCHAR(100));

CREATE TABLE order_items (
  product_no INTEGER REFERENCES products,
  order_id INTEGER REFERENCES orders,
  quantity INTEGER NOT NULL,
  PRIMARY KEY (product_no, order_id),
  CHECK (quantity > 0));
```

(a) A relational database schema containing three tables.

```
1) INSERT INTO products(product_no, name, price, discounted_price) VALUES(0, 'ijyv', 280, 1);   ✓
2) INSERT INTO orders(order_id, shipping_address) VALUES(0, 'kt');                               ✓
3) INSERT INTO order_items(product_no, order_id, quantity) VALUES(0, 0, 290);                    ✓
4) INSERT INTO products(product_no, name, price, discounted_price) VALUES(1, '', 728, 299);      ✓
5) INSERT INTO orders(order_id, shipping_address) VALUES(-285, 'shpalcrku');                     ✓
6) INSERT INTO order_items(product_no, order_id, quantity) VALUES(0, 0, 1);                      ✗
```

(b) An example test case automatically generated and consists of INSERT statements for a database specified by the relational schema in part (a). The test case exercises the PRIMARY KEY of the order_items table as false. Normally inspected by a tester who is checking schema correctness, the ✓ and ✗ marks denote whether or not the data contained within each INSERT satisfied the schema's integrity constraints and was accepted into the database.



(c) The vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ representation used by random and fitness-guided search techniques for finding the test data for each INSERT forming the test in part (b).

Figure 3.2: The *Products* relational database schema and an example test case.

While value copying and randomisation may "fix" a part of the test data vector for a particular integrity constraint, it may also invalidate some other part. For example, ensuring the distinctness of a primary key value, through RANDOMIZENONMATCHES, may destroy its foreign key reference, previously set through COPYMATCHES. To handle this concern, the functions are applied one after the other in a loop, continuing until an overall solution is found or resources (i.e., a given number of algorithm iterations) are exhausted.

Now every function in DOM-RND's main loop will be discussed to show how it generates test data for the *Products* schema in Figure 3.2 and for satisfying/violating each of the different types of integrity constraints.

Primary Keys and "Unique" Constraints

The functions COPYMATCHES and RANDOMIZENONMATCHES work to ensure that values in `INSERT` statements pertaining to primary keys/`UNIQUE` constraints are (a) *distinct* when such constraints need to be satisfied, else ensuring those values are (b) *identical* should the constraint need to be violated. Ensuring distinctness is not usually difficult to achieve by selecting values randomly, as the probability of choosing the same value more than once is small. Nevertheless, if two values match in the vector, the second value is regenerated by RANDOMIZENONMATCHES. Alternatively, if a primary key/`UNIQUE` constraint is required to be violated by the test case, the values for the columns involved in the latter, constraint-violating, `INSERT` statement are copied from an earlier `INSERT` statement to the same table appearing in the test case. For example, the `PRIMARY KEY` of the `order_items` table is required to be violated, that is the test case of Figure 3.2. Therefore, v_{16} and v_{17} is required to be equal to v_7 and v_8, respectively. Thus, the COPYMATCHES copies v_{16} and v_{17}'s values from v_7 and v_8. If there is a choice of subsequent `INSERT` statements from which to copy a value, COPYMATCHES selects one at uniform random. If the primary key/unique constraint involves multiple columns, then multiple values are copied together from a selected prior `INSERT` statement in the test case.

Foreign Keys

Compared to the previously described functions, COPYMATCHES and randomizeNonMatches work in a reverse fashion for foreign keys in that the constraint is *satisfied* when values match for the relevant columns across `INSERT` statements in the test case, and *violated* when they do not. As with the previous two functions, RANDOMIZENONMATCHES generates non-matching values randomly, while COPYMATCHES copies values that are supposed to match from elsewhere in the vector. Take the example of `INSERT`s 3 and 6 from the test of Figure 3.2b and the values of `product_no` and `order_id`, which individually need to match the corresponding column in the `products` and `orders` table. In both cases, two options exist. For `product_no`, a matching value is found in `INSERT` statements 1 and 4 (i.e., v_1 and v_{10} in the vector). For `order_no`, a matching value is found in `INSERT` statements 2 and 5 (i.e., v_5 and v_{14}). As before, where choices exist, COPYMATCHES selects one at uniform random.

"Not Null" Constraints

Depending on the coverage requirement, the SETORREMOVENULLS function works to overwrite values in the vector with a random value where a non-`NULL` value is required (e.g., to satisfy a `NOT NULL` constraint), and copies `NULL` into the vector where a `NULL` value is required instead (e.g., to violate a `NOT NULL` constraint). For instance, to violate the `NOT NULL` constraint on the `name` column of the `products` table, the SETORREMOVENULLS function would replace the value of either v_2 or v_{11} with a `NULL` value.

"Check" Constraints

As they involve arbitrary predicates that need to be solved, `CHECK` constraints cannot generally be satisfied nor violated by copying values from elsewhere in the vector. The SOLVECHECKCONSTRAINTS function generate random values, (e.g., for `price` and `discounted_price` in the `products` table). This is the default approach taken by DOM-RND, and the one employed unless otherwise specified. Values are chosen at random from the domain of the column type, or from the pool of constants mined from the schema (i.e., the mechanism described for the Random$^+$ method, introduced in Section 2.3.5). The latter mechanism is particularly useful for constraints of the form "`CHECK a IN (x, y, z)`" where the column `a` has to be set to one of "`x`", "`y`", or "`z`" to be satisfied. These values are hard to "guess" randomly without any prior knowledge, yet since the values "`x`", "`y`", or "`z`" will have been added to the constant pool, DOM-RND is able to select and use them as test data values.

3.4 DOMINO-RANDOM Empirical Evaluation

The aim of this section's empirical evaluation is to determine if DOM-RND will improve the efficiency and effectiveness of test data generation for relational database schemas. That is, improving the test data generation coverage, timing, and fault-finding capabilities. Therefore, the study is designed to answer these two research questions:

RQ1: Test Suite Generation for Coverage—Effectiveness and Efficiency. How effective is DOM-RND at generating high-coverage tests for database integrity constraints and how fast does it do so, compared to the state-of-the-art AVM?

RQ2: Fault-Finding Effectiveness of the Generated Test Suites. How effective are the test suites generated by DOM-RND in regard to fault-finding effectiveness, and how do they compare to those generated by the state-of-the-art AVM?

3.4.1 Methodology

Techniques

To answer the RQs, DOM-RND will be empirically evaluated, comparing it to the AVM. Both variants of the AVM. The first was studied by McMinn et al. [7], as discussed in Section 3.2 and Section 2.3.5, and uses default values for the first initialisation of the vector (and then random re-initialisation following each restart), which is referred as “AVM-D”. For a better comparison with DOM-RND, the variant of AVM where all initialisations are performed randomly, which is called “AVM-R”, will be evaluated. And only performing Random$^+$ to obtain its coverage levels to establish a baseline for which to compare all techniques.

Subject Schemas

the experiments were performed by using the 34 relational database schemas listed Appendix A. In order to answer RQ1, and to generate test suites with which to assess fault-finding capability, a coverage criterion is required. For this purpose, the combination of three coverage criteria were adopted: “ClauseAICC”, “AUCC”, and “ANCC”, as introduced in Section 2.3.5. The reason for using this combined coverage criterion is that it was reported as the strongest to find seeded faults [7], combining the capability to find faults of both commission and omission.

The set of 34 relational database schemas were featured in previous work on testing database schemas (e.g., [16, 7, 136]). Since Houkjær et al. noted that complex real-world relational schemas often include features such as composite keys and multi-column foreign-key relationships [86], the schemas chosen for this study reflect a diverse set of features, from simple instances of integrity constraints to more complex examples involving many-column foreign key relationships. The number of tables in each relational database schema varies from 1 to 42, with a range of just 3 columns in the smallest schemas, to 309 in the largest. Some schemas are examples from many sources, and they are simpler than some other schemas used in this study, they nevertheless proved challenging for database analysis tools such as the DBMonster data generator [16].

DBMSs

The HyperSQL, PostgreSQL, and SQLite DBMSs hosted the subject schemas. Each of these database management systems is supported by our *Schema-*

Analyst tool [14]; they were chosen for their performance differences and varying design goals. PostgreSQL is a full-featured, extensible, and scalable DBMS, while HyperSQL is a lightweight, small DBMS with an "in-memory" mode that avoids disk writing. SQLite is a lightweight DBMS that differs in its interpretation of the SQL standard in subtly different ways from HyperSQL and PostgreSQL. A wide variety of real-world programs, from different application domains, use these three DBMSs.

RQ1

For RQ1, each test data generation method ran on each schema and DBMS, for each coverage requirement. Each technique moves onto the next requirement (or terminating if all requirements have been considered) if test data has been successfully found, or after iterating 100,000 times if it has not. Obtaining the coverage levels, and the test data generation time, for 30 repetitions of each method with each of the 3 database schemas and the 3 DBMSs.

RQ2

For RQ2, the fault-finding strength were studied on each generated est suite for RQ1, following standard experimental protocols that use mutation analysis [137]. Adopting Wright et al.'s procedure [101], using the same set of mutation operators that mutate the schema's integrity constraints, for more details please refer to Section 2.3.7. These operators add, remove, and swap columns in primary key, `UNIQUE`, and foreign key constraints, while also inverting `NOT NULL` constraints and manipulating the conditions of `CHECK` constraints. RQ2 deems the automatically generated test suites to be effective if they can "kill" a mutant by distinguishing between it and the original schema, leading to the formulation of the higher-is-better mutation score as the ratio between the number of killed and total mutants [138, 139, 140].

Experimentation Environment

All the experiments were performed on a dedicated Ubuntu 14.04 workstation, with a 3.13.0–44 GNU/Linux 64-bit kernel, a quad-core 2.4GHz CPU, and 12GB of RAM. All input (i.e., schemas) and output (i.e., data files) were stored on the workstation's local disk. Using the default configurations of PostgreSQL 9.3.5, HyperSQL 2.2.8, and SQLite 3.8.2, with HyperSQL and SQLite operating with their "in-memory" mode enabled.

Statistical Analysis

Using four tables, this chapter reports the mean values for the 30 sets of evaluation metrics (i.e., coverage values, time to generate test suites in seconds, and mutation scores) obtained for each schema with each DBMS. For reasons similar to those of Poulding and Clark [141], the means were reported instead of medians: for data that was sometimes bi-modal, the median value was one of the "peaks" while the mean reported a more useful statistic between the peaks.

Using statistical significance and effect size, we further compared DOM-RND pairwise with every other studied technique. Following Arcuri and Briand recommendations regarding randomisation algorithms, we performed the non-parametric Mann-Whitney U test for statistical significance [142]. Performing one-sided tests (sided for each technique in each pairwise comparison) with p-value < 0.01 regarded as significant. In all of the results tables, the technique's value is marked if it was significant, using the "$\triangledown$" symbol if the mean result is lower compared to DOM-RND or the "$\vartriangle$" symbol if the mean result is higher compared to DOM-RND. In addition to significance tests, the effect sizes are calculated using the non-parametric $\hat{A}$ metric of Vargha and Delaney [143]. Classifying an effect size as "large" if $|\hat{A} - 0.5| > 0.21$. In all of the tables, the technique's result marked with the "$*$" symbol if DOM-RND performed significantly better and with a large effect size.

Threats to Validity

External Validity. The diverse nature of real software makes it impossible for me to claim that the studied schemas are representative of all types of relational database schemas. Therefore, we attempted to select diverse schemas that came from both open-source and commercial software systems, choosing from those used in past studies [7]. Also, the results may not generalise to other DBMSs. However, HyperSQL, PostgreSQL, and SQLite are three widely used DBMSs with contrasting characteristics—and they also implement key aspects of the SQL standard related to defining schemas with various integrity constraints.

Internal Validity. To control threats of both the stochastic behaviour of the techniques and the possibility of operating system events interfering with the timings, we repeated the experiments 30 times. To mitigate threats associated with the statistical analysis we (a) used non-parametric statistical tests and (b) performed all the calculations with the R programming language, writing unit tests to check the results.

Construct Validity. It is worth noting that, while this chapter does not report the cost of running the generated tests, they normally consist of a few `INSERT`s whose cost is negligible and thus not of practical significance.

3.4.2 Experimental Results

RQ1: Test Suite Generation for Coverage—Effectiveness and Efficiency

Table 3.1 shows the mean coverage scores for DOM-RND (written as DR) compared to the two AVM variants and Random$^+$ (written as R$^+$). In the table, a value annotated with the "$\triangledown$" symbol means that significance tests reveal that a technique obtained a significantly lower coverage score than DOM-RND (written as DR), while "$\triangle$" means the technique obtained a significantly higher coverage than DOM-RND. The poor results for Random$^+$ underscore that test data generation is not a trivial task for most schemas, except for *NistDML183* and *NistXTS748*. Random$^+$ is outperformed by every other method. Note that while the table only reports statistical significance and a large effect size for DOM-RND pairwise with every other technique, the coverage scores for the two versions of the AVM are also significantly better with a large effect size in each case when compared to Random$^+$. Since it is dominated by the three other methods, from hereon we will discount

Table 3.1: Mean Coverage Scores For Each Technique

	HyperSQL				**PostgreSQL**				**SQLite**			
Schema	DR	AVM-R	AVM-D	R$^+$	DR	AVM-R	AVM-D	R$^+$	DR	AVM-R	AVM-D	R$^+$
ArtistSimilarity	100	100	100	*▼59	100	100	100	*▼59	100	100	100	*▼62
ArtistTerm	100	100	100	*▼60	100	100	100	*▼60	100	100	100	*▼63
BankAccount	100	100	100	*▼85	100	100	100	*▼85	100	100	100	*▼87
BookTown	99	99	99	*▼92	99	99	99	*▼92	99	99	99	*▼92
BrowserCookies	100	*▼99	100	*▼58	100	*▼99	100	*▼58	100	*▼99	100	*▼59
Cloc	100	100	100	*▼92	100	100	100	*▼92	100	100	100	*▼92
CoffeeOrders	100	100	100	*▼58	100	100	100	*▼58	100	100	100	*▼62
CustomerOrder	100	100	100	*▼42	100	100	100	*▼42	100	100	100	*▼42
DellStore	100	100	100	*▼93	100	100	100	*▼93	100	100	100	*▼93
Employee	100	100	100	*▼89	100	100	100	*▼89	100	100	100	*▼90
Examination	100	100	100	*▼83	100	100	100	*▼83	100	100	100	*▼84
Flights	100	*▼97	100	*▼59	100	*▼97	100	*▼59	100	*▼97	100	*▼58
FrenchTowns	100	100	100	*▼35	100	100	100	*▼35	100	100	100	*▼35
Inventory	100	100	100	*▼96	100	100	100	*▼96	100	100	100	*▼96
Iso3166	100	100	100	*▼85	100	100	100	*▼85	100	100	100	*▼89
IsoFlav_R2	100	100	100	*▼88	100	100	100	*▼88	100	100	100	*▼88
iTrust	100	100	100	*▼92	100	100	100	*▼92	100	100	100	*▼92
JWhoisServer	100	100	100	*▼86	100	100	100	*▼86	100	100	100	*▼87
MozillaExtensions	100	100	100	*▼88	100	100	100	*▼88	100	100	100	*▼88
MozillaPermissions	100	100	100	*▼96	100	100	100	*▼96	100	100	100	*▼96
NistDML181	100	100	100	*▼64	100	100	100	*▼64	100	100	100	*▼65
NistDML182	100	100	100	*▼62	100	100	100	*▼62	100	100	100	*▼65
NistDML183	100	100	100	100	100	100	100	100	100	100	100	100
NistWeather	100	100	100	*▼57	100	100	100	*▼57	100	100	100	*▼75
NistXTS748	100	100	100	100	100	100	100	100	100	100	100	100
NistXTS749	100	100	100	*▼86	100	100	100	*▼86	100	100	100	*▼86
Person	100	100	100	*▼93	100	100	100	*▼93	100	100	100	*▼94
Products	98	98	98	*▼70	98	98	98	*▼70	98	98	98	*▼79
RiskIt	100	100	100	*▼68	100	100	100	*▼68	100	100	100	*▼70
StackOverflow	100	100	100	*▼96	100	100	100	*▼96	100	100	100	*▼96
StudentResidence	100	100	100	*▼70	100	100	100	*▼70	100	100	100	*▼74
UnixUsage	100	100	100	*▼50	100	100	100	*▼50	100	100	100	*▼52
Usda	100	100	100	*▼90	100	100	100	*▼90	100	100	100	*▼90
WordNet	100	100	100	*▼90	100	100	100	*▼90	100	100	100	*▼89

Random$^+$ as a comparison technique for generating test suites for database schemas.

The state-of-the-art AVM-D obtains 100% coverage for each schema, except for *BookTown* and *Products*, which contain infeasible coverage requirements. DOM-RND matches this effectiveness (it cannot do any better, but it does not any worse either), while AVM-R has difficulties with *BrowserCookies* and *Flights*. For these schemas, AVM-R has trouble escaping a local optimum for a particular coverage requirement. It restarts many times, but fails to find test data before its resources are exhausted. The use of default values always provides a good starting point for AVM-D to cover the requirements concerned, and as such, it does not suffer from these problems.

DOM-RND does not use a fitness function, and so does not face this issue.

Thus, for coverage scores, DOM-RND performs identically to AVM-D, but better than AVM-R for some schemas, and significantly better than Random$^+$ for all non-trivial schemas.

Table 3.2: Mean Test Generation Times (in seconds)

	HyperSQL			PostgreSQL			SQLite		
Schema	DR	AVM-R	AVM-D	DR	AVM-R	AVM-D	DR	AVM-R	AVM-D
ArtistSimilarity	0.49	*Δ0.96	*Δ0.60	1.02	*Δ1.41	*Δ1.08	0.29	*Δ0.72	*Δ0.44
ArtistTerm	0.56	*Δ1.15	*Δ0.72	2.60	*Δ3.10	*Δ2.68	0.33	*Δ0.91	*Δ0.54
BankAccount	0.53	*Δ0.83	*Δ0.76	1.33	*Δ1.62	*Δ1.59	0.32	*Δ0.62	*Δ0.57
BookTown	1.03	*Δ1.41	*Δ1.09	7.18	*Δ7.54	7.24	0.57	*Δ0.95	*Δ0.64
BrowserCookies	0.66	*Δ5.76	*Δ3.37	3.22	*Δ8.19	*Δ5.85	0.42	*Δ5.97	*Δ3.23
Cloc	0.51	*Δ0.63	*Δ0.60	1.15	*Δ1.28	*Δ1.19	0.30	*Δ0.41	*Δ0.43
CoffeeOrders	0.65	*Δ1.11	*Δ1.08	4.43	*Δ4.90	*Δ4.74	0.40	*Δ0.85	*Δ0.82
CustomerOrder	0.86	*Δ3.36	*Δ1.87	7.94	*Δ10.62	*Δ8.65	0.55	*Δ3.22	*Δ1.79
DellStore	0.83	*Δ1.63	*Δ1.56	4.19	*Δ4.96	*Δ4.84	0.48	*Δ1.28	*Δ1.14
Employee	0.55	*Δ0.82	*Δ0.90	1.05	*Δ1.27	*Δ1.34	0.34	*Δ0.59	*Δ0.70
Examination	0.78	*Δ1.74	*Δ1.57	4.05	*Δ4.94	*Δ4.84	0.49	*Δ1.45	*Δ1.27
Flights	0.69	*Δ4.93	*Δ3.99	2.48	*Δ6.59	*Δ5.77	0.45	*Δ5.23	*Δ3.90
FrenchTowns	0.68	*Δ1.94	*Δ1.70	3.02	*Δ4.17	*Δ3.86	0.43	*Δ1.63	*Δ1.94
Inventory	0.48	*Δ0.56	*Δ0.60	0.70	*Δ0.75	*Δ0.80	0.28	*Δ0.35	*Δ0.44
Iso3166	0.47	*Δ0.55	*Δ0.55	0.48	*Δ0.54	*Δ0.50	0.27	*Δ0.35	*Δ0.40
IsoFlav_R2	0.75	*Δ1.31	*Δ1.27	5.13	*Δ5.69	*Δ5.48	0.43	*Δ0.99	*Δ0.93
iTrust	4.91	*Δ47.91	*Δ15.99	46.95	*Δ85.67	*Δ55.28	4.58	*Δ47.11	*Δ14.12
JWhoisServer	0.89	*Δ2.09	*Δ1.88	4.03	*Δ5.15	*Δ4.87	0.55	*Δ1.79	*Δ1.55
MozillaExtensions	0.86	*Δ2.01	*Δ1.92	6.36	*Δ7.62	*Δ7.34	0.55	*Δ1.65	*Δ1.55
MozillaPermissions	0.51	*Δ0.61	*Δ0.66	1.08	*Δ1.16	*Δ1.19	0.31	*Δ0.40	*Δ0.49
NistDML181	0.53	*Δ0.83	*Δ0.71	1.55	*Δ1.80	*Δ1.71	0.32	*Δ0.62	*Δ0.54
NistDML182	0.76	*Δ2.36	*Δ1.94	5.74	*Δ7.43	*Δ6.81	0.50	*Δ2.10	*Δ2.09
NistDML183	0.51	*Δ0.58	*Δ0.64	1.32	*Δ1.44	*Δ1.44	0.30	*Δ0.36	*Δ0.48
NistWeather	0.71	*Δ1.42	*Δ1.31	1.93	*Δ2.64	*Δ2.52	0.48	*Δ1.14	*Δ1.22
NistXTS748	0.48	*Δ0.53	*Δ0.61	0.61	*Δ0.66	*Δ0.71	0.28	*Δ0.33	*Δ0.50
NistXTS749	0.55	*Δ0.78	*Δ0.82	1.54	*Δ1.81	*Δ1.77	0.33	*Δ0.57	*Δ0.69
Person	0.55	*Δ1.05	*Δ1.60	0.68	*Δ1.17	*Δ1.73	0.34	*Δ0.87	*Δ1.56
Products	0.71	*Δ1.72	*Δ1.71	2.30	*Δ3.28	*Δ3.40	0.47	*Δ1.33	*Δ1.38
RiskIt	1.00	*Δ3.62	*Δ2.31	11.70	*Δ14.72	*Δ12.53	0.63	*Δ3.48	*Δ1.99
StackOverflow	0.82	*Δ1.17	*Δ1.47	4.66	*Δ4.83	*Δ5.01	0.48	*Δ0.84	*Δ1.12
StudentResidence	0.59	*Δ0.97	*Δ0.78	1.43	*Δ1.72	*Δ1.54	0.38	*Δ0.75	*Δ0.63
UnixUsage	0.87	*Δ3.48	*Δ1.93	11.11	*Δ13.31	*Δ11.52	0.52	*Δ2.99	*Δ1.67
Usda	0.86	*Δ1.40	*Δ1.53	6.23	*Δ6.40	*Δ6.47	0.49	*Δ1.01	*Δ1.03
WordNet	0.68	*Δ0.97	*Δ1.13	3.64	*Δ3.92	*Δ3.99	0.40	*Δ0.67	*Δ0.84

Table 3.2 gives the mean times for each technique to obtain the coverage scores in Table 3.1, excluding Random$^+$. In the table, a value annotated with a "∇" symbol means that significance tests reveal that a technique required a significantly shorter time than DOM-RND, while "Δ" indicates the technique needed a significantly longer time than DOM-RND. The results show that DOM-RND outperforms both of the AVM variants, which incur significantly

higher times in each case, with a large effect size. The difference is most noticeable for larger schemas (i.e., *iTrust* and *BrowserCookies*). With *iTrust*, DOM-RND is approximately 40 seconds faster than AVM-R with each of the DBMSs, representing a speedup of 8–10 times for HyperSQL and SQLite. Compared to AVM-D, DOM-RND is approximately 10 seconds faster for each DBMS. For smaller schemas, the differences are significant but less pronounced. Although DOM-RND is faster than the AVM variants for these schemas, the practical difference is almost negligible.

Concluding RQ1, DOM-RND yields the same coverage scores as the state-of-the-art AVM-D, but in less time. Compared to DOM-RND, AVM-R is slower and has slightly worse coverage.

RQ2: Fault-Finding Effectiveness of the Generated Test Suites

Table 3.3 shows the mean mutation scores obtained by each technique's generated test suites. The results show that DOM-RND achieved significantly higher mutation scores (i.e., values annotated with a "$\triangledown$" symbol) than the state-of-the-art AVM-D technique for 20–23 of the 34 schemas, depending on the DBMS, with a large effect size in almost every case. AVM-R is more competitive with DOM-RND, however. For these two techniques there are fewer differences in effectiveness. Therefore, it seems that developing test cases from a random starting point is important for mutation killing effectiveness. AVM-D starts from the same default values, which may remain unchanged, depending on the test requirement. Ultimately, there is less diversity across this method's test suites, leading them to kill fewer mutants.

Variations in DOM-RND's effectiveness compared to AVM-R stem from differences in the approach taken for generating test data: DOM-RND always copies values where it can for certain types of requirement and integrity constraint, whereas AVM-R may legitimately opt to use `NULL` instead of a matching value. For instance, DOM-RND satisfies foreign keys with `NULL` values, unless there are `NOT NULL` constraints on the columns of the key. The occasional use of `NULL` leads AVM-R to kill more mutants than DOM-RND for some schemas, and fewer for others. The relative advantages depend on the

Table 3.3: Mean Mutation Scores

	HyperSQL			**PostgreSQL**			**SQLite**		
Schema	DR	AVM-R	AVM-D	DR	AVM-R	AVM-D	DR	AVM-R	AVM-D
ArtistSimilarity	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
ArtistTerm	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
BankAccount	95.9	95.5	*▼88.5	95.9	95.5	*▼88.5	96.4	96.1	*▼86.7
BookTown	99.5	99.4	*▼97.6	99.5	99.4	*▼97.6	99.1	99.0	*▼85.5
BrowserCookies	96.3	▼95.6	*▼92.3	96.3	▼95.6	*▼92.3	95.9	96.1	*▼86.5
Cloc	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
CoffeeOrders	100.0	100.0	100.0	100.0	100.0	100.0	98.6	*△100.0	*▼94.6
CustomerOrder	97.5	97.5	*▼94.0	97.5	97.4	*▼93.9	98.0	98.0	*▼95.2
DellStore	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Employee	97.7	97.6	*▼95.3	97.7	97.6	*▼95.3	97.3	97.4	*▼84.1
Examination	100.0	▼99.8	*▼97.3	100.0	▼99.8	*▼97.3	99.2	99.6	*▼85.8
Flights	99.8	*▼97.9	*▼95.2	99.8	*▼97.9	*▼95.2	100.0	*▼98.2	*▼84.3
FrenchTowns	94.3	94.3	*▼82.5	94.3	94.3	*▼82.5	94.6	94.6	*▼83.3
Inventory	100.0	100.0	*▼87.5	100.0	100.0	*▼88.2	100.0	100.0	*▼75.0
Iso3166	99.6	99.6	*▼77.8	99.6	99.6	*▼77.8	99.7	99.7	*▼80.0
IsoFlav_R2	99.7	99.8	*▼87.0	99.7	99.8	*▼87.0	99.7	99.8	*▼84.4
iTrust	99.7	*▼99.6	*▼95.8	99.7	*▼99.6	*▼95.8	99.2	99.2	*▼83.6
JWhoisServer	99.6	99.6	*▼78.7	99.6	99.6	*▼78.7	99.6	99.5	*▼76.6
MozillaExtensions	99.8	99.6	*▼82.1	99.8	99.6	*▼82.1	99.7	99.5	*▼71.3
MozillaPermissions	100.0	100.0	100.0	100.0	100.0	100.0	99.8	99.8	*▼76.7
NistDML181	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistDML182	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistDML183	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
NistWeather	98.2	△100.0	*▼93.3	98.2	△100.0	*▼93.3	98.4	△100.0	*▼93.8
NistXTS748	93.3	93.7	*▼88.2	93.3	93.7	*▼88.2	92.9	93.3	*▼87.5
NistXTS749	95.0	95.0	95.0	95.0	95.0	95.0	91.7	*△96.0	92.0
Person	97.8	96.5	*▼81.0	97.8	96.5	*▼81.0	98.8	97.3	*▼81.8
Products	87.2	87.1	▼86.5	87.2	87.1	▼86.5	87.8	87.7	▼87.1
RiskIt	100.0	100.0	*▼99.5	100.0	100.0	*▼99.5	99.5	*△99.9	*▼89.3
StackOverflow	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
StudentResidence	97.2	▼96.5	*▼94.4	97.2	▼96.5	*▼94.4	95.7	96.6	*▼87.2
UnixUsage	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	*▼98.2
Usda	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
WordNet	97.8	97.6	*▼93.7	97.8	97.6	*▼93.7	98.5	97.9	*▼87.4

DBMS: For HyperSQL and PostgreSQL, DOM-RND obtains a significantly higher mutation score for five schemas, while AVM-R performs better for one schema. While some of these comparisons are accompanied by a large effect size, the differences in means are usually marginal. Conversely, for the SQLite DBMS, DOM-RND is better for two schemas, while AVM-R is better for three. This is likely because SQLite allows the use of `NULL` values in primary key columns, giving more opportunity for `NULL` to be used as a data value in tests for schemas that it hosts. AVM-R can exploit this opportunity by using `NULL` whereas DOM-RND does not—in turn leading to more times for which using `NULL` can result in the killing of a mutant.

For nine schemas, a 100% mutation score was achieved regardless of technique and DBMS. Closer inspection revealed that these schemas had few or simple constraints (i.e., all `NOT NULL` constraints), the mutants of which were easy to kill.

The schemas with the weakest mutation score was *Products*, with a maximum of 87.8% with DOM-RND and the SQLite DBMS. Closer inspection revealed that this schema had many live mutants generated as a result of `CHECK` constraints, thus motivating the hybrid DOMINO-AVM investigated in RQ3.

To conclude for RQ2, the results show that DOM-RND is more effective at killing mutants than the state-of-the-art AVM-D technique. The results reveal few differences in the mutation score of DOM-RND compared to AVM-R. Yet, RQ1 showed that DOM-RND generates data significantly faster than AVM-R—with marginally better coverage as well—and therefore is the most effective *and* efficient technique of the three.

3.5 The Hybrid DOMINO-AVM Method

DOM-RND does not solve `CHECK` constraints with domain-specific heuristics, as with other types of constraint, and lead to a weaker detection of `CHECK` constraint mutants. Instead, its random method relies on a solution being "guessed" without any guidance. Thus, presenting a hybrid version of DOM-RND, called "DOMINO-AVM", that uses the AVM to handle this aspect of the test data generation problem. The AVM uses the fitness function that would have been employed in the pure AVM version of Section 3.2, providing guidance to the required values that may be valuable when the constraints are complex and difficult to solve by chance selection of values. This is illustrated in Figure 3.3 where the role of the SOLVECHECKCONSTRAINTSWITHAVM function is to run the AVM on the `CHECK` constraint and generate data with guidance.

3.5.1 DOMINO-AVM Empirical Evaluation

In this section aims to empirically evaluate if the DOMINO-AVM will improve the efficiency and effectiveness of test data generation for relational database

```
1 $RANDOMIZE(\vec{v})$
2 while $\neg$ termination criterion do
3     copyMatches($\vec{v}, r$)
4     randomizeNonMatches($\vec{v}, r$)
5     setOrRemoveNulls($\vec{v}, r$)
6     solveCheckConstraintsWithAVM($\vec{v}, r$)
7 end
```

Figure 3.3: The Domino-AVM algorithm that automatically generate, according to some coverage criterion r, a vector $\vec{v}$ of variables appearing in the `INSERT` statements of a test case for database schema integrity constraints.

schemas. Therefore, the same methodology in previous section was used to answer the following research question:

RQ3: The Hybrid Domino-AVM Technique. How do test suites generated by Domino-AVM compare to DOM-RND's in terms of efficiency, coverage, and fault-finding capability?

To answer RQ3, coverage was measured, the time taken to obtain coverage, and the mutation score of the Domino-AVM's tests for the schemas with `CHECK` constraints (i.e., those for which the Domino-AVM, which uses the AVM instead of random search to solve `CHECK` constraints, will register a difference). These results were compared to those of DOM-RND, which uses the default mode of random search to solve `CHECK` constraints.

RQ3: The Hybrid Domino-AVM Technique

For the schemas with `CHECK` constraints — that is, the schemas for which Domino-AVM could potentially improve upon DOM-RND — Table 3.4 reports the mean results of coverage, test suite generation time, and mutation scores. For ease of comparison, DOM-RND results are re-reported for these schemas alongside those obtained for Domino-AVM.

Domino-AVM achieves full coverage for all schemas, except for those that involve infeasible test requirements, as did DOM-RND. Perhaps surprisingly, however, Domino-AVM is generally no better in terms of time to generate the test suites, and is in fact reported as significantly worse in

Table 3.4: Mean Results of DOMINO-AVM Compared to DOM-RND for Subject Schemas That Have `CHECK` constraints

A value annotated with a "▼" symbol means that significance tests reveal that DOM-RND (DR) obtained a significantly lower result than DOMINO-AVM (DOMINO-AVM), while "∆" shows that DOM-RND obtained a significantly higher result than DOMINO-AVM. A "*" symbol indicates that the effect size was large when comparing the technique with DOM-RND.

	Coverage						**Timing**						**Mutation Score**					
	HyperSQL		**PostgreSQL**		**SQLite**		**HyperSQL**		**PostgreSQL**		**SQLite**		**HyperSQL**		**PostgreSQL**		**SQLite**	
Schema	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR	DOMINO-AVM	DR
BookTown	99.3	99.3	99.3	99.3	99.3	99.3	1.08	*▼1.03	7.25	7.18	0.61	*▼0.57	99.4	99.5	99.4	99.5	99.1	99.1
BrowserCookies	100.0	100.0	100.0	100.0	100.0	100.0	0.69	*▼0.66	3.21	3.22	0.44	*▼0.42	95.9	96.3	95.9	96.3	96.0	95.9
CustomerOrder	100.0	100.0	100.0	100.0	100.0	100.0	0.91	*▼0.86	8.13	*▼7.94	0.60	*▼0.55	97.4	97.5	97.4	97.5	98.0	98.0
Employee	100.0	100.0	100.0	100.0	100.0	100.0	0.58	*▼0.55	1.08	1.05	0.36	*▼0.34	97.0	∆97.7	97.0	∆97.7	96.7	97.3
Examination	100.0	100.0	100.0	100.0	100.0	100.0	0.82	*▼0.78	4.14	4.05	0.52	*▼0.49	99.6	∆100.0	99.6	∆100.0	99.2	99.2
Flights	100.0	100.0	100.0	100.0	100.0	100.0	0.77	*▼0.69	2.59	*▼2.48	0.51	*▼0.45	100.0	99.8	100.0	99.8	100.0	100.0
iTrust	100.0	100.0	100.0	100.0	100.0	100.0	4.82	∆4.91	46.46	46.95	4.54	4.58	99.6	*∆99.7	99.6	*∆99.7	99.1	99.2
NistWeather	100.0	100.0	100.0	100.0	100.0	100.0	0.72	0.71	2.00	▼1.93	0.49	0.48	99.7	▼98.2	99.7	▼98.2	99.9	▼98.4
NistXTS748	100.0	100.0	100.0	100.0	100.0	100.0	0.49	*▼0.48	0.61	0.61	0.29	*▼0.28	93.3	93.3	93.3	93.3	92.9	92.9
NistXTS749	100.0	100.0	100.0	100.0	100.0	100.0	0.56	*▼0.55	1.61	1.54	0.34	*▼0.33	95.0	95.0	95.0	95.0	92.0	91.7
Person	100.0	100.0	100.0	100.0	100.0	100.0	0.74	*▼0.55	0.85	*▼0.68	0.51	*▼0.34	97.0	97.8	97.0	97.8	97.0	∆98.8
Products	97.9	97.9	97.9	97.9	98.1	98.1	0.75	*▼0.71	2.38	▼2.30	0.50	*▼0.47	87.1	87.2	87.1	87.2	87.7	87.8
StudentResidence	100.0	100.0	100.0	100.0	100.0	100.0	0.58	0.59	1.38	1.43	0.36	*∆0.38	96.6	∆97.2	96.6	∆97.2	95.0	95.7

the table for several schemas, with an accompanying large effect size. This indicates that, for this study's schemas, random search can successfully solve the `CHECK` constraints and utilisation of the constant pool mined it from the schema. Thus, using the AVM is of no additional benefit in terms of speeding up the test data generation process.

In terms of mutation score, there is one schema (i.e., *NistWeather*) where DOMINO-AVM is significantly better than DOM-RND for all DBMSs, and cases where the reverse is true (e.g., *Employee* and *Examination*) but for the HyperSQL and PostgreSQL DBMSs only. The actual differences in means are small, and are accounted for by the random solver's use of constants mined from the schema with DOM-RND, as opposed to the search-based approach taken by DOMINO-AVM. In the cases where DOM-RND does better, it is for relational constraints where a value is being compared to a constant (e.g., `x >= 0`). The use of the seeded constant (i.e., `0` for `x`) means that a boundary value is being used, which helps to kill the mutants representing a changed relational operator (e.g., from `>=` to `>`).

On the other hand, DOMINO-AVM may use any value that satisfies the constraint (e.g., `1` for `x`), according to the fitness function, that may not fall on the boundary and not kill the mutant. Conversely, not using constant seeding can help to kill other mutant types, which is what happens with *NistWeather*. Here, DOM-RND only satisfies a `CHECK` constraint by using a value mined from the schema, leading to a repetition of the same value across different `INSERT` statements of a test case. In contrast, the fitness function gives guidance to different values that satisfy the `CHECK` constraint for DOMINO-AVM. This increased diversity helps DOMINO-AVM to consistently kill an additional mutant that DOM-RND was unable to kill.

The conclusion for RQ3 is that the AVM's potential to improve the generation of data for test requirements involving `CHECK` constraints is only of benefit for a few cases. The use of random search, as employed by DOM-RND, achieves similar results to DOMINO-AVM in a shorter amount of time.

Overall Results Conclusions: The results indicate that DOM-RND is the best method, achieving the highest mutation scores (RQ2) and requiring the least time to generate test suites (RQ1). The coverage it obtains is optimal and is comparable with the previous state-of-the-art-technique, AVM-D. Yet, it generates test data that is more diverse, which has a positive impact on the fault-finding capability of its test suites. Given that DOM-RND handles `CHECK` constraints randomly, while the AVM is fitness-guided, a hybrid technique would seem fruitful. However, the results from RQ3 contradict this intuition. Instead, it seems that AVM's superiority over random search, as shown by the results for RQ1, is to do with generating test data for other types of integrity constraint. For the studied schemas, test data can be effectively generated for `CHECK` constraints with a random method — although DOMINO-AVM does generate tests that are better at killing mutants for one particular subject.

3.6 Summary

Since databases are a valuable asset protected by a schema, this chapter introduced DOM-RND, a method for automatically generating test data that systematically exercises the integrity constraints in relational database schemas. Prior ways to automate this task (e.g., [16, 7]) adopted search-based approaches relying on the Alternating Variable Method (AVM). Even though DOM-RND is more efficient than the AVM, its domain-specific operators enable it to create tests that match the coverage of those produced by this state-of-the-art method.

DOM-RND can also generate tests that are better at killing mutants than AVM-D, a version of the AVM that starts the search from a set of default values (e.g., '0' for integers or the empty string for strings). This is advantageous because the test data values generated by DOM-RND, not being based on default values, have greater diversity. Following this insight, we also studied an AVM that starts with random values. Experiments show that, while AVM-R has a similar mutant killing capability to DOM-RND, its overall coverage scores are not as high as the presented method's and it takes significantly longer to generate its tests. Finally, we compared DOM-RND to a hybridisation combining the domain-specific operators with the use of

AVM for the CHECKs, finding that this alternative is less efficient that the presented method and no more effective.

Since prior work has shown the importance of human-readable test data [70, 69], the following chapter will study whether testers understand DOM-RND's data values. That is, evaluating the generated test data (i.e., inputs) and trying to identify which data contribute positively or negatively on understanding the tests within the evaluation phase (i.e., helping to decrease the human oracle cost).

Chapter 4

What Factors Make SQL Test Cases Understandable For Testers?

The content of this chapter is based on the published work during this PhD, and presented in the International Conference on Software Maintenance and Evolution (ICSME) 2019 [144].

4.1 Introduction

The previous chapter showed that domain-specifies operators incorporated with a random search, called DOMINO, significantly improved the effectiveness and efficiency of test data generation. While testers need to act as an oracle and understand each test case, DOMINO generates random values that can difficult to understand. Therefore, this chapter will investigate understandability factors associated with database schema testing.

It is challenging to create test cases that are understandable and maintainable [18, 19] — mainly when the tests use complex and inter-dependent `INSERT` statements to populate a relational database [20]. While automated test data generators can create test cases that aid systematic database schema testing [16], the human cost associated with inspecting test output and understanding test outcomes is often overlooked [21].

When database schemas evolve [22], their automatically generated tests should be understandable by humans. Source code understandability is subjective, with developers having different views of automatically generated tests [23]. For example, if testers are deciding whether or not the database will reject a test, some may prefer English-like strings, while others may appreciate simple values such as empty strings. Yet, it is crucial to create understandable database schema tests, since comprehensible test inputs support human comprehension of test outcomes and may expedite the process of finding and fixing faults [24].

Intending to identify the factors that make SQL tests understandable for human testers, this chapter uses several automated test data generation methods to create tests for database schemas. Therefore, four techniques are implemented and categorised according to the data that they generate: (1) random values; (2) default values that use empty strings for characters and constants for numeric values; (3) values from a language model used by Afshan et al. [69], combined with a search-based technique, Alternating Variable Method (AVM); and (4) reused values derived from either column names or a library of readable values. A human study is conducted to evaluate the understandability of the data generated. The human participants were tasked with explaining test outcomes for data arising from the five data generators (i.e., inspecting the test behaviour). Therefore, participants were asked to identify which `INSERT` statement, if any, would be rejected by the database because it violated a schema's integrity constraint.

This chapter highlights two key findings. The first is that the data values in `INSERT`s influence human understandability: using default values for elements not involved in the test — but necessary for adhering to SQL's syntax rules — aided participants, allowing them to identify and understand the critical values easily. Yet, negative numbers and "garbage" strings hindered a human's ability to reason about the rejection of `INSERT` statements. The second finding is more far-reaching and in confirmation of prevailing wisdom among database developers: humans found the outcome of tests challenging to predict when `NULL` was used in conjunction with foreign keys and `CHECK` constraints. Even though `NULL`s limit test understandability for humans, this result suggests that `NULL` use in tests can surface the confusing semantics of database schemas.

Overall, this chapter makes the following contributions:

1. New test data generator variants that are adapted into *SchemaAnalyst* that aims to improve readability (Section 4.3.1).
2. A human study that assesses the understandability of automatically generated test data by using a realistic task in which participants must determine which `INSERT`, if any, would be rejected by a relational database (Sections 4.3 – 4.4).
3. Readability guidelines for schema tests, derived from quantitative and qualitative feedback from industrial and academic experts in the human study, directing both manual testers and creators of automated testing tools (Sections 4.5 – 4.6).

To support the replication of this chapter's experimental results and to facilitate the testing of relational database schemas, the proposed techniques are implemented into *SchemaAnalyst* [145] and the procedures into scripts. Replication instructions are available in Appendix B.2.

4.2 Motivation

To motivate this chapter, the *BrowserCookies* schema that was used in Chapter 2 is iterated as an example in Figure 4.1(a). Figure 4.1(b) gives examples of tests, produced by DOM-RND and AVM-D, that violate the `UNIQUE` constraint of the `cookies` table. Both AVM-D and DOM-RND assume an empty database, building up the sequence of `INSERT`s required to first populate the database with valid values, so that the constraint can be tested with identical values for the columns focused on by the final `INSERT` of each test. The sequence of statements also involves inserting data into the `places` table so that the foreign key of the `cookies` table is not violated instead of the `UNIQUE` constraint, which is the ultimate target of this test case.

Automated test data generators can help testers to avoid the tedious and error-prone task of manually writing tests for a database schema. Also, the previous chapter and in prior [7, 101, 105], the techniques automatically generated tests that can effectively cover the schema and detect synthetic schema faults.

```
CREATE TABLE places (          CREATE TABLE cookies (
  host TEXT NOT NULL,            id INTEGER PRIMARY KEY NOT NULL,
  path TEXT NOT NULL,            name TEXT NOT NULL,
  title TEXT,                    value TEXT,
  visit_count INTEGER,           expiry INTEGER,
  fav_icon_url TEXT,             last_accessed INTEGER,
  PRIMARY KEY(host, path)        creation_time INTEGER,
);                               host TEXT,
                                 path TEXT,
                                 UNIQUE(name, host, path),
                                 FOREIGN KEY(host, path) REFERENCES places(host, path),
                                 CHECK (expiry = 0 OR expiry > last_accessed),
                                 CHECK (last_accessed >= creation_time),
                               );
```

(a) The *BrowserCookies* relational database schema

AVM-D

```
1) INSERT INTO places(host, path, title, visit_count, fav_icon_url)
   VALUES ('', '', '', 0, '')
2) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
   VALUES (0, '', '', 0, 0, 0, '', '')
3) INSERT INTO places(host, path, title,visit_count, fav_icon_url)
   VALUES ('a', '', '', 0, '')
4) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
   VALUES (1, '', '', 0, 0, 0, '', '')
```

DOM-RND

```
1) INSERT INTO places(host, path, title, visit_count, fav_icon_url)
   VALUES ('xuksiu', 'fwkjy', 'bmmniu', -53, 'f')
2) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
   VALUES (0, 'iywt', 'ryl', 0, -357, -877, 'xuksiu', 'fwkjy')
3) INSERT INTO places(host, path, title,visit_count, fav_icon_url)
   VALUES ('lmm', 'j', 'w', 907, NULL)
4) INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)
   VALUES (131, 'iywt', 'mdofmfl', NULL, NULL, 106, 'xuksiu', 'fwkjy')
```

(b) Automatically generated test cases using AVM-D and DOM-RND that violates a `UNIQUE` constraint,

Figure 4.1: The *BrowserCookies* relational database schema with examples of automatically generated test case data.

That is one example of a test case from a test suite can have many test cases. For example, a basic coverage criterion that simply satisfies and violates each constraint would therefore have 20 test requirements, The more complex combination of ClauseAICC, ANCC and AUCC, with higher fault revealing power [7], has 71. Therefore, testers must still act as an "oracle" for a test when they judge whether it passed or failed [24], a challenging task that is often overlooked.

The effort expended by a human acting as an oracle for a test suite — that is, understanding each test case and its outcomes, reasoning about whether a test should pass or fail and whether the observed behaviour is correct or otherwise — is referred to as the “human oracle cost” [56]. Human oracle costs can be categorised either quantitative or qualitative. It is possible to decrease the quantitative costs by, for instance, reducing the number of tests in a suite or the length of the individual tests. Strategies to reduce the qualitative costs often involve modifying the test data generators so that they create values that are more meaningful to human testers [69, 70]. With the ultimate goal of reducing human oracle costs, this chapter identifies the factors that influence test understandability.

Although human oracle costs can be ameliorated by creating automated test data generation methods that consider readability (please refer to Section 2.4 for more details), to the best of my knowledge there is no prior work aiming to characterise and limit the qualitative human oracle costs associated with the automated testing of a database schema. As a first step, we must determine how generated test data affects a human’s understanding of a test’s behaviour. Thus, before focusing on generating tests that limit human oracle costs, it is prudent to identify the characteristics that make test cases easy for testers to understand and reason about. This chapter reports on a human study performing this important task.

As an example, even though each of the tests in Figure 4.1(b) successfully violate the intended `UNIQUE`, they employ different values because they were created with the two previously described automated test data generation techniques. Depending on the generated test data, it may be more or less challenging for a tester to effectively reason about test outcomes [24] and determine whether or not the tests achieved the goal of creating inputs that do not satisfy an integrity constraint. For instance, the second and fourth `INSERT` statements from AVM-D assign empty strings for the values of the `UNIQUE` constraint, while the second and fourth `INSERT`s from the DOM-RND technique use randomly generated strings. Since every data generator works differently, each created test may have varying values — all of which may differ in their human understandability and support of effective testing — for both those attributes involved in testing an integrity constraint and the other schema attributes.

Knowing that the readability of test inputs influences test case understandability [24], variants of AVM and DOMINO are created that generate more readable data values. This enables the characterisation of factors involved in the comprehension of tests for relational database schemas, which is the focus of this chapter's study, the design of which the next section describes.

4.3 Methodology

In order to act as a human oracle, testers must understand the behaviour of a test. The aim of this study is to find out what properties of relational schema tests, comprising SQL `INSERT`s, make them easy for humans to understand.

I have studied five different ways to automatically generate tests, based on the two main techniques, AVM and DOMINO, as introduced in the previous chapter. Each technique embodies a different strategy for producing the test inputs (i.e., the values within the `INSERT`s) that may affect the human comprehension of those tests. These involve the use of default values, random values, pre-prepared data such as dictionary words, or data specifically generated to have English-like qualities.

4.3.1 Automated Test Case Generation Techniques

Figure 4.2 introduce each automated method with example test cases for the *NistWeather* schema. The test cases generated by each method, featured in part (b) of this figure, aim to satisfy the `CHECK` constraint on the `MONTH` column of the `Stats` table, starting from an initially empty database. In order to insert a valid row in the `Stats` table, a row must first be inserted into the `Station` table, thereby ensuring that the foreign key declared in the `Stats` table is not violated. Thus, each test case consists of two `INSERT` statements.

The first two test data generators, AVM-D and AVM-LM, are based on the Alternating Variable Method from prior chapters.

AVM-D previously introduced in previous chapter and was chosen for this study as it has featured in a number of prior papers devoted to testing relational database integrity constraints (e.g., [16, 7]). An example test case generated by AVM-D is shown in Figure 4.2(b). The default values — empty strings and zeros — are shown in each `INSERT` statement and are used when AVM-D did not need to modify the data values to fulfil the test requirement. The values of `1` and `127` are needed to satisfy the `CHECK` constraints on `MONTH` and `TEMP_F`, respectively.

```
CREATE TABLE Station (
  ID INTEGER PRIMARY KEY,
  CITY VARCHAR(20),
  STATE CHAR(2),
  LAT_N INTEGER NOT NULL,
  LONG_W INTEGER NOT NULL,
  CHECK (LAT_N BETWEEN 0 and 90),
  CHECK (LONG_W BETWEEN 180 AND -180)
);
```

```
CREATE TABLE Stats (
  ID INTEGER REFERENCES STATION(ID),
  MONTH INTEGER NOT NULL
  TEMP_F INTEGER NOT NULL,
  RAIN_I INTEGER NOT NULL,
  CHECK (MONTH BETWEEN 1 AND 12),
  CHECK (TEMP_F BETWEEN 80 AND 150),
  CHECK (RAIN_I BETWEEN 0 AND 100),
  PRIMARY KEY (ID, MONTH)
);
```

(a) The *NistWeather* relational database schema.

AVM-D	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W)`
		`VALUES (0, '', '', 0, 0);`

	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I)`
		`VALUES (0, 1, 127, 0);`

AVM-LM	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W)`
		`VALUES (100, 'Thino', 'jo', 0, 0);`

	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I)`
		`VALUES (100, 6, 127, 1);`

DOM-RND	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W)`
		`VALUES (100, 'ivjyv', 'jr', 0, 0);`

	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I)`
		`VALUES (100, 12, 90, 40);`

DOM-COL	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W)`
		`VALUES (100, 'CITY_0', 'ST', 2, 0);`

	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I)`
		`VALUES (100, 12, 90, 1);`

DOM-READ	1)	`INSERT INTO Station(ID, CITY, STATE, LAT_N, LONG_W)`
		`VALUES (100, 'sidekick', 'ba', 90, 150);`

	2)	`INSERT INTO Stats(ID, MONTH, TEMP_F, RAIN_I)`
		`VALUES (100, 12, 80, 12);`

(b) Generated test cases with multiple techniques that satisfies a `CHECK` constraint for column `MONTH`.

Figure 4.2: The *NistWeather* relational database schema with examples of automatically generated test case data.

AVM-LM is the basic AVM algorithm but with an additional post-processing step. Following the generation of data using the AVM (this time, starting with random, rather than default values), the strings in a test case are optimised for "English-likeness" using a language model, similar to that employed by Afshan et al. [69]. This method replaces every instance of a string in each `INSERT` statement of a test case with a new string generated using the language model. The algorithm generates 10,000 strings of the same length and picks the one with the best language model score. We included AVM-LM because, in Afshan et al.'s study of automated test data generation for C programs, the incorporation of a language model as an extra fitness component in the search-based method helped to produce more readable strings that made tests easier and quicker for human testers to understand [69]. Figure 4.2(b) shows an example of a test case generated with AVM-LM. The test case does not use default values, but rather starts with a sequence of data values that are either randomly generated or randomly selected from constants used in the schema itself. This method creates English-like words for test strings (i.e., "`Thino`" and "`jo`").

The next three methods, DOM-RND, DOM-COL, and DOM-READ are variants of the DOMINO from Section 4.2.

DOM-RND was also chosen for this study as it was featured in the previous chapter's study and found to obtain the highest mutation scores out of all studied testing methods. Also, DOM-RND generate values that are nearly identical to a random test data generator, and from an understandability perspective its allowing me to not include a purely random test data generator. Figure 4.2(b) gives an example of a test in which this method generated all values randomly or randomly selected from constants mined from the schema.

DOM-COL is a variant of DOMINO that, instead of using a randomly generated value for a string, uses the value's associated column name with a sequential integer suffix. The motivation behind DOM-COL is the intuition that, if a data value embodies the column name, testers should easily match data values in an `INSERT` with their columns. Since this is only viable with strings, for integer data DOM-COL attempts to use sequentially generated integers instead of random values. DOM-COL's example test in Figure 4.2(b) shows how "`CITY_0`" is used as one of the values. Since the `STATE` column has a two character limit, the chosen value is a random subsequence of the column name, which here is the first two characters.

DOM-READ is another variant of DOMINO that selects values from a database that is used separately from the testing process and is populated for the schema. The motivation for this customisation of DOMINO is similar to that of AVM-LM: readable values from an existing database should make test data values easier to follow in the `INSERT` statements that contain them. For the purposes of this study, the databases populator is used that is a Java library called DataFactory [146], which fills string fields with English words. The test for DOM-READ in Figure 4.2(b) is similar to that of DOM-RND's except that it features either English words or word-like subsequences for length-constrained fields (i.e., "`sidekick`" and "`ba`").

While AVM-D and DOM-RND have previously appeared in both the literature [7] and Chapter 3, AVM-LM, DOM-COL, and DOM-READ are new techniques designed for this study of test input comprehension.

4.3.2 Measuring Comprehension

Program comprehension, the task of reading and understanding programs, is a complex cognitive task [147]. It often involves understanding a system's behaviour through the development of either general-purpose or application-specific software knowledge [148]. This chapter uses multiple-choice questions to measure this human knowledge and identify comprehension factors. While some studies use multiple-choice questions to assess problem-solving skill [149], others report that performance on a multiple-choice quiz correlates with knowledge of a written text [150]. In comparison to open-ended short-answer essays, multiple-choice questions are normally more reliable because they constrain the responses [151]. Overall, this prior work shows that multiple-choice questions can surface a human's understanding and problem-solving skills.

4.3.3 Research Questions

With the goal of identifying the factors that make SQL test cases understandable, we designed a human study to focus on answering the following two research questions:

RQ1: Success Rate in Comprehending the Test Cases. How successful are testers at correctly comprehending the behaviour of schema test cases generated by automated techniques?

RQ2: Factors Involved in Test Case Comprehension. What are the factors of automatically generated SQL `INSERT` statements that make them easy for testers to understand?

4.3.4 Experimental Set-up

Schemas and Generators. To generate tests, the publicly available *SchemaAnalyst* tool [14] is used, which already provides an implementation of the AVM-D and DOM-RND techniques for database schema testing. The DOM-COL and DOM-READ (and their value-initialising libraries) and AVM-LM (and its language model) are added to *SchemaAnalyst*, making the enhanced tool, as shown in Figure 4.3, available for download at *SchemaAnalyst* GitHub[1] repository. Using *SchemaAnalyst*, tests are generated for the *BrowserCookies* schema in Figure 4.1 and *NistWeather* schema in Figure 4.2, applying each of the five test generation techniques. These database schemas are selected because, taken together, they have the five main types of integrity constraint (i.e., primary keys, foreign keys, `CHECK`, `NOT NULL`, and `UNIQUE`) and different data types (e.g., integers, text, and constrained strings).

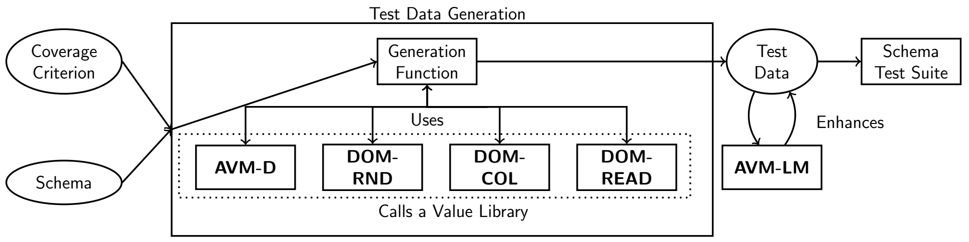


Figure 4.3: The inputs and outputs of the enhanced *SchemaAnalyst* tool.

Test Cases. The *SchemaAnalyst* was configured to generate test suites by fulfilling a coverage criterion that produces tests that exercise each integrity constraint of the schema with `INSERT` statements that are (a) accepted, because the test data in the `INSERT` statements *satisfies* the integrity constraint along with any other constraints that co-exist in the same table, and (b) contains an `INSERT` statement that is rejected, because test data in it *violates* the integrity constraint (while satisfying all other constraints) [7]. We selected

[1]https://github.com/schemaanalyst/schemaanalyst

one example of a test that satisfies each different type of integrity constraint (e.g., primary keys and foreign keys) and one example of a test case that violates each type of integrity constraint for each relational schema.

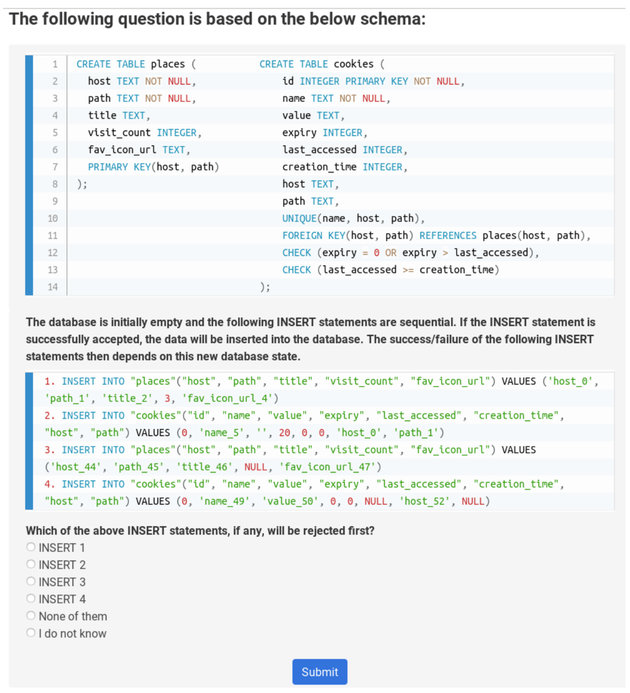
When there were multiple test cases to choose from (because, for example, the schema involves multiple `CHECK` constraints), we selected one at random. *BrowserCookies* involves at least one of each of the main five types of integrity constraint, while *NistWeather* involves all the main types of integrity constraint except a `UNIQUE`. As such, the set of test cases used for the questionnaire consisted of ten test cases for *BrowserCookies* and a further eight for *NistWeather* — to satisfy and violate each of the integrity constraint types — generated by each of the five techniques, resulting in a total of 90 test cases overall. *SchemaAnalyst* was configured to generate test cases suitable for database schemas hosted by the PostgreSQL DBMS. PostgreSQL was selected as its behaviour is generally accepted as closest to the SQL standard [152, 153].

I then incorporated the generated test cases in a comprehension task delivered by a web-based questionnaire system, as further described in Section 4.3.5.

Pilot Trial. The number of questions, test cases, and schemas were carefully chosen using a pilot trial. The trial revealed that when participants were given more than two schemas they got confused and could not remember schema properties, which is not realistic. I also noted that humans completed the tasks in less than an hour when given tests covering all of the integrity constraints in a schema like the ones in Figures 4.1 and 4.2.

4.3.5 Design of the Human Study

Web-Based Questionnaire Supporting Two Studies. I created a web application to allow human participants to answer questions about the automatically generated test cases. Each question has its own individual web page featuring a specific test. Figure 4.4 gives a screenshot of the system that shows the schema for which *SchemaAnalyst* generated the test case at the top of the page, with the `INSERT` statements making up the test underneath.



The following question is based on the below schema:

```
CREATE TABLE places (                CREATE TABLE cookies (
  host TEXT NOT NULL,                    id INTEGER PRIMARY KEY NOT NULL,
  path TEXT NOT NULL,                    name TEXT NOT NULL,
  title TEXT,                            value TEXT,
  visit_count INTEGER,                   expiry INTEGER,
  fav_icon_url TEXT,                     last_accessed INTEGER,
  PRIMARY KEY(host, path)                creation_time INTEGER,
);                                       host TEXT,
                                         path TEXT,
                                         UNIQUE(name, host, path),
                                         FOREIGN KEY(host, path) REFERENCES places(host, path),
                                         CHECK (expiry = 0 OR expiry > last_accessed),
                                         CHECK (last_accessed >= creation_time)
                                     );
```

The database is initially empty and the following INSERT statements are sequential. If the INSERT statement is successfully accepted, the data will be inserted into the database. The success/failure of the following INSERT statements then depends on this new database state.

```
1. INSERT INTO "places"("host", "path", "title", "visit_count", "fav_icon_url") VALUES ('host_0',
'path_1', 'title_2', 3, 'fav_icon_url_4')
2. INSERT INTO "cookies"("id", "name", "value", "expiry", "last_accessed", "creation_time",
"host", "path") VALUES (0, 'name_5', '', 20, 0, 0, 'host_0', 'path_1')
3. INSERT INTO "places"("host", "path", "title", "visit_count", "fav_icon_url") VALUES
('host_44', 'path_45', 'title_46', NULL, 'fav_icon_url_47')
4. INSERT INTO "cookies"("id", "name", "value", "expiry", "last_accessed", "creation_time",
"host", "path") VALUES (0, 'name_49', 'value_50', 0, 0, NULL, 'host_52', NULL)
```

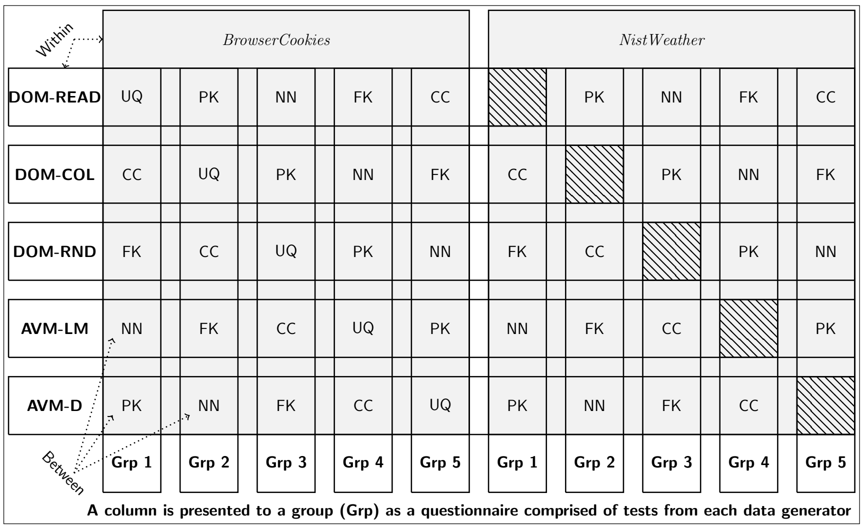
Which of the above INSERT statements, if any, will be rejected first?

○ INSERT 1
○ INSERT 2
○ INSERT 3
○ INSERT 4
○ None of them
○ I do not know

Figure 4.4: Screenshot showing how the survey system displays a question.

The questionnaire then required the participants to select the first `INSERT` statement of the test case, if any, that is rejected by the DBMS because it violates one of more integrity constraints of the schema. If the test is designed to satisfy all of the integrity constraints, none of the `INSERT` statements will fail, whereby participants should select “None of them”. The goal is for participants to focus on the test inputs, acting as oracles for these tests that do not have assertion statements. When a participant could not decide on the answer, a “I don’t know” option was provided for them to select, thereby preventing them from having to select a response at random to continue to the next question. Importantly, adding the “I don’t know” option helped to prevent guessing from influencing the results.

To answer the RQs, we designed a human study based on this questionnaire. In the first part, referred to as the “silent” study, participants answered the questionnaire under “exam conditions” (i.e., they were not allowed to interrupt other participants or confer). This allowed me to obtain a relatively large set of quantitative data from the questionnaire in a short amount of time. The second part took the form of a "think aloud" study in which we collected more detailed and qualitative information from a smaller number of participants. The participants did not receive the correct answers to any of the questions, which might have influenced their answers to questions involving later test cases. Importantly, this type of mixed design is often used to validate quantitative results [154].



	BrowserCookies					*NistWeather*				
DOM-READ	UQ	PK	NN	FK	CC	▨	PK	NN	FK	CC
DOM-COL	CC	UQ	PK	NN	FK	CC	▨	PK	NN	FK
DOM-RND	FK	CC	UQ	PK	NN	FK	CC	▨	PK	NN
AVM-LM	NN	FK	CC	UQ	PK	NN	FK	CC	▨	PK
AVM-D	PK	NN	FK	CC	UQ	PK	NN	FK	CC	▨
	Grp 1	**Grp 2**	**Grp 3**	**Grp 4**	**Grp 5**	**Grp 1**	**Grp 2**	**Grp 3**	**Grp 4**	**Grp 5**

Within
Between
A column is presented to a group (Grp) as a questionnaire comprised of tests from each data generator

Figure 4.5: The mixed study design with two within-subjects variables (i.e., a schema and a data generator) and one between-subjects variable (i.e., a test case).

The Silent Study (SS). Designed to answer RQ1, this study involved 25 participants recruited from the student body at the University of Sheffield, studying Computer Science (or a related degree) at either the undergraduate or PhD level. As part of the recruitment and sign-up process, potential participants completed an assessment in which they had to say whether four `INSERT` statements would be accepted or rejected for a table with three constraints. Participates were not invited if they got more than one answer wrong, ensuring that we included capable participants with adequate SQL knowledge. The web-based questionnaire asked the level of SQL experience of each participant, which varied between less than a year for nine participants to over five years for two. I designed this quiz to focus on the understandability of test inputs and not the understandability of basic SQL commands.

I assigned each participant to one of five groups randomly, such that there were five participants in each group. The study had two within-subject variables (i.e., the database schemas and the test case generation techniques) and one between-subject variable (i.e., the specific test cases themselves) [155], as shown in Figure 4.5. In the figure, each test case represents a question and is denoted by an integrity constraint's test (i.e., a Primary Key ("PK"), Foreign Key ("FK"), `UNIQUE` constraint (UQ), `NOT NULL` constraint (NN), or `CHECK` constraint (CC)). The hashed box shows that this schema did not have a `UNIQUE` test. This show all groups answered questions involving an adequate test case created by each test generation technique for each of the two schemas and a specific integrity constraint. I also assigned a test made by a generator to precisely one group, resulting in five responses per test. Since each cell in Figure 4.5 represents a separate test for satisfying and violating each constraint, this means that there were 450 data points in total, with 250 for *BrowserCookies* and 200 for *NistWeather*. Although two questions were added at the start of a question set so that participants could practice and get familiar with each schema, we did not analyse the responses to these questions. Each participant was financially compensated with £10, encouraging them to do their best to understand the schema tests and complete the questionnaire in under an hour.

The Think Aloud Study (TAS). This study was also designed to answer RQ2, recruiting five new individuals to complete the questionnaire, assigning each to their own group and allowing full coverage of the questions in the questionnaire. Participants were asked to say their thought processes aloud, a technique commonly used in the HCI research community for studying human cognitive processes in problem-solving tasks [156]. This protocol allows for the inferences, reasons, and decisions made by participants to be surfaced when they complete an assignment [157]. We performed this study, prompting participants to say "why" they had chosen an answer if they had not already verbalised their reasoning.

Audio recording of each participant's session were made and manually transcribing it to text afterwards. Following this, all of these statements are analysed. When at least three of the five participants said the same thing, this chapter reports it as a "key observation" in the answer to RQ2.

The five participants comprised three additional Computer Science PhD students from the University of Sheffield and two industry participants who each had two years of experience. With these five participants, I restricted myself to prompting them with a "why?" question to get them to reveal their thought processes, without any further interactions. A sixth participant was recruited and performed the TAS in a randomly assigned group. In contrast to the first five participants, the sixth participant was asked direct questions inspired by comments that others made. This sixth participant was a developer from a large multi-national corporation with over 10 years of software development experience, including with the SQL. As such, this participant is referred to as the "experienced industry engineer" and was not counted among the official TAS participants. Instead, RQ2 answer uses the expert as an additional source of comments and reported alongside the first five participants.

4.3.6 Threats to Validity

External Validity. The selection of schemas for this chapter's study is a validity threat because those chosen may yield results that are not be evident for real schemas. To mitigate this threat, two schemas are selected that feature all the integrity constraints and data types commonly evident in schemas [22]. Since they may not represent those often used in practice, the tests used in the study are also a validity threat. To address this matter, an open-source automated test data generation tool was used, *SchemaAnalyst* [14], and configured to create effective tests according to a recommended adequacy criterion [7]. This decision guaranteed that the study's participants considered tests that can exercise all of a schema's integrity constraints as both *true* and *false*. The use of a few relational schemas and tests is also a validity threat. It is worth noting that we purposefully limited the number of these artefacts to ensure that participants could complete the questionnaire in a reasonable amount of time, thereby mitigating the potentially negative effects of fatigue. Since no previous human studies have been done in this area and the categorising of test comprehension factors, this study therefore considered the first and began with a small-scale experiment using a small number of participants. Given the relatively small number of total data points, a statistical power calculation was used to see the percentage chance of detecting differences in the human responses to the questionnaire.

Internal Validity. The potential for a learning effect is a validity threat that could arise when participants become better at answering questions as the questionnaire progresses, due to their experience with prior tasks. This threat was mitigated with randomising the presentation order for questions and schemas. The "think aloud" (TAS) experiment also had threats that we attempted to mitigate. To ensure that all study participants had a uniform experience, the people in the TAS had to abide by a restricted form of interaction with me, ensuring that they did not inappropriately discover facets of the comprehension task. Since participants in a think aloud may be naturally reluctant to verbalise their thought process, they are instructed to "stream" their thoughts during their completion of the questionnaire. Another potential validity threat is that the majority of the participants in the studies were students. However, the TAS included two industrialists and an expert who had technique rankings that were similar to those arising from the silent studies with the students. This trend suggests that it is acceptable to use students to identify the factors that make SQL tests understandable, in broad confirmation of prior results in software engineering [158].

Construct Validity. The measurement of a subjective concept like understandability is also a validity threat. To assess test understandability, we determined how successful human testers were at identifying which `INSERT` statement, if any, would be rejected by the database because it violated an integrity constraint — a viable proxy to understandability that we could accurately calculate. Yet, a study of this nature raises other concerns since participants might not be accustomed to using the questionnaire application to determine the outcome of a SQL test case. It is also possible that testers might have better knowledge of a database schema that they designed. To overcome both of these final concerns, the study included two practice questions with responses that were not recorded.

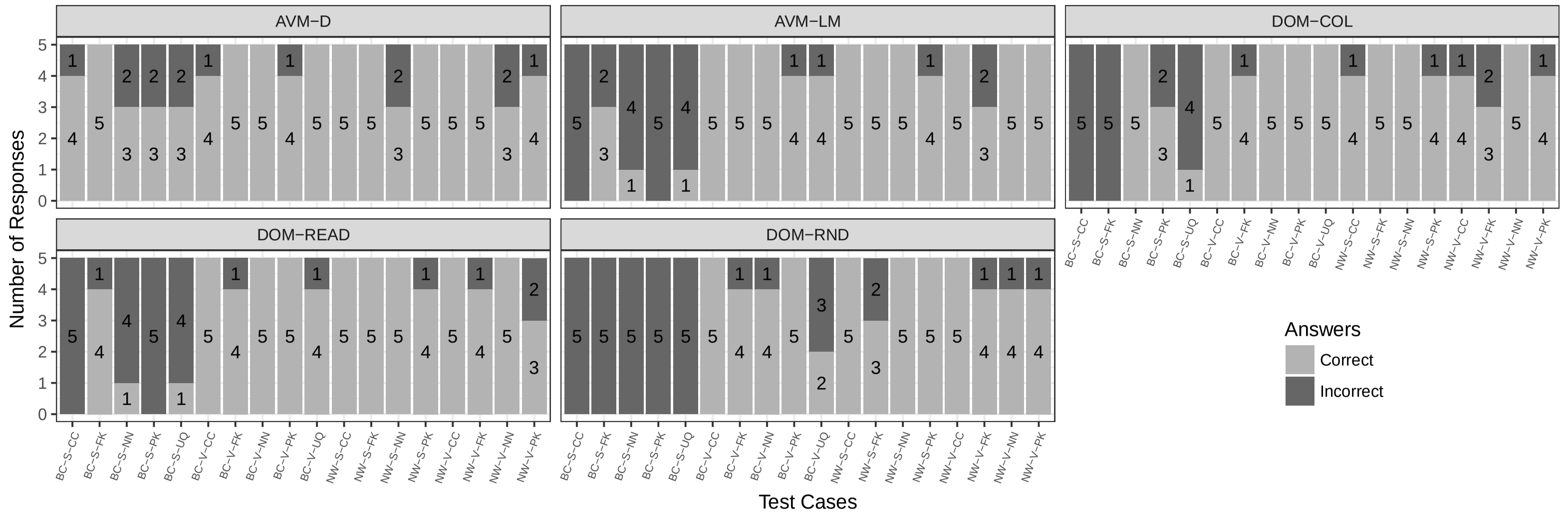


Figure 4.6: A stacked bar plot that shows the count of correct and incorrect responses for each automated test generation technique and test case.

4.4 Answers to the Research Questions

RQ1: Success Rate in Comprehending the Test Cases. Table 4.1 shows the number of correct and incorrect responses for RQ1. A response is correct if a participant successfully selected the first `INSERT` that was rejected by the DBMS, or the "None" option, if all the `INSERT` statements are accepted. The "I do not know" option was not selected by participants in response to any of the questions in the silent study (SS).

Table 4.1: Correct and Incorrect Answers for the Silent Study

Technique	Correct Responses	Incorrect Responses	Percentage Correct	Rank
AVM-D	76	14	84%	1
AVM-LM	65	25	72%	=3
DOM-COL	67	23	74%	2
DOM-RND	55	35	61%	5
DOM-READ	65	25	72%	=3

Tests generated by AVM-D were most easily comprehended: participants correctly responded 84% of the time. Conversely, tests produced by DOM-RND were the most misunderstood: participants only correctly responded 61% of the time for this method. AVM-LM, DOM-COL, and DOM-READ, which all employ operations to produce more readable strings, achieved similar numbers of correct responses between 72 and 74%.

The Fisher Exact test was performed on the results on each pair of techniques, which revealed a statistically significant difference between AVM-D and DOM-RND, with a p-value < 0.001. However, at the same alpha-level of 0.05, there were no statistically significant differences between the other techniques. A post-hoc test called "Power of Fisher's Exact Test for Comparing Proportions" was also used to compute the statistical power of Fisher's Exact test [159]. This test shows that, with 90 responses each for DOM-RND and AVM-D, there will be a 93% chance of detecting a significant difference at the 0.05 significance level, assuming that the response score is 84% and 61% for AVM-D and DOM-RND, respectively. For the other test data generators, a post-hoc test calculates that there is a 50% or less chance of detecting a significant difference, suggesting the need for more human participants.

Figure 4.6 shows the numbers of correct and incorrect responses for each test case. Each stacked bar of the plot corresponds to a specific test case. The horizontal-axis labels designate which schema the test case was generated for (either *BrowserCookies*, denoted by the "BC" prefix, or *NistWeather*, denoted by the "NW" prefix). These are suffixed by the test case type – either the satisfaction ("S") or violation ("V") of a specific integrity constraint (a primary key ("PK"), foreign key ("FK"), `UNIQUE` constraint (UQ), `NOT NULL` constraint (NN), or `CHECK` constraint (CC)). This plot reveals that participants had particular trouble with DOM-RND and identifying test cases where there was no rejected `INSERT` statement for the *BrowserCookies* schema, as shown in the figure by the bars labelled with the "BC-S-" prefix. These are test cases designed to exercise an integrity constraint such that all data in the `INSERT` statements is successfully entered into the database. All of the questions involving these test cases were answered incorrectly for DOM-RND. Similarly, participants struggled with these types of test cases for AVM-LM, DOM-COL, and DOM-READ: they correctly answered 5, 9, and 6 questions out of 25, respectively. However, for AVM-D, participants did not encounter the same issues, answering 18 out of 25 questions correctly. The ratio of correct/incorrect answers is more or less similarly evenly distributed for other test types, although even for these remaining types of tests, DOM-RND remains the weakest performer in terms of correct responses.

In conclusion for RQ1, the silent study showed that participants seem to most easily comprehend the behaviour of the test cases generated by AVM-D, as evidenced by the fact that they answered the most questions correctly for test cases generated by this technique. In contrast, the most difficult test cases to understand were those generated by DOM-RND. The other techniques, that fall in between these two extremes, have a similar influence on the human comprehension of schema tests.

A Think-aloud study was designed with the aim of finding out more about these potential differences in the minds of the human participants, the results of which is discuss next.

RQ2: Factors Involved in Test Case Comprehension. The TAS resulted in fewer overall responses as there were only five participants. Yet, Table 4.2 shows the recorded answers follow a similar pattern to those given by participants for RQ1: AVM-D produces tests that are understood the best, with DOM-RND the worst, AVM-LM and DOM-COL falling between the two, and DOM-READ tying AVM-D in this study. The main purpose of the TAS was to surface what participants thought about the tests for which they answered questions. There were seven key observations (KOs) made by three or more of the five participants, each of which is discuss next.

Table 4.2: Correct and Incorrect Answers for the Think Aloud Study

Technique	Correct Responses	Incorrect Responses	Percentage Correct	Rank
AVM-D	16	2	89%	=1
AVM-LM	14	4	78%	4
DOM-COL	15	3	83%	3
DOM-RND	12	6	67%	5
DOM-READ	16	2	89%	=1

■ *Confusing Behaviour of Foreign Keys* (KO1) *and* `CHECK` *Constraints* (KO2) *with* `NULL`. When `NULL` is used on columns without `NOT NULL` constraints but with other integrity constraints, participants tended to think that the `INSERT` statement should be rejected. All five stated this for foreign keys, while four commented they thought this was true of `CHECK` constraints. Yet, this is not the behaviour defined by the SQL standard [160].

One participant admitted that they "think it is easier to just look at the ones that have a `NULL` to see if they are rejected first". While it was easy for the participants to spot `NULL`s, they found it confusing to judge how they would behave when interacting with the schema's other integrity constraints. For example, one participant stated that "the `path` [a `FOREIGN KEY` column in the *BrowserCookies* schema] is `NULL` which is not going to work, so I will stop thinking there and judge [`INSERT` statement] four to be the faulty statement." Another participant said that a "`CHECK` constraint should be a `NOT NULL` by default" even when the constraint involved columns that could be `NULL`.

The experienced industry engineer stated the following when he encountered a `NULL` on a `FOREIGN KEY` column: "the schema does not allow it" and on another question that "it should fail the `FOREIGN KEY` because of the `NULL` [in one of the compound foreign key columns] and the fact that value does not exist [the other foreign key column value in the referenced table]". He also debated with himself on the issue of whether `CHECK` constraints should not allow a `NULL` as he was "not sure about the boolean logic around `NULL`s — I do not think `NULL` is equal to zero and I do not think `NULL` is greater than `NULL`". He asked himself "can I treat a `NULL` as zero?". After answering the question with "I do not know", I asked him "Do you think `NULL`s in `CHECK` constraints are a bit confusing?". He answered "Yes, I am very wary with `NULL`". After completing the survey, he made the following observation: "In a work situation, I would have looked up how `NULL` is interpreted in a logical constraint. I did not find them hard to read but I do not know how the DBMS is going to interpret a `NULL`".

To conclude this KO, `NULL` is confusing for testers, and the frequency of its use in tests is a factor for comprehension.

■ *Negative Numbers Require More Comprehension Effort When Used in* `CHECK` *Constraints* (KO3). Negative numbers confused four participants when the column is numeric and used within a boolean logic of a `CHECK`. Participants repetitively checked negative numbers when they were compared together. A participant reported that negative numbers were more difficult than positive numbers because "it takes more time to do mental arithmetic" when they are in comparisons. Another participant said negative numbers "are not realistic".

The experienced industry engineer also commented on negative numbers when he was prompted after answering a survey question with them. He stated "they are harder, slightly, to think about but it is OK and I can reason about them". For negative numbers with primary keys he said: "It feels that you would not use a negative value on a primary key".

To conclude this KO, the use of negative numbers increases the comprehension effort for database schema test cases.

■ *Randomly Generated Strings Require More Comprehension Effort to Com-*

pare (KO4). Four think-aloud participants said that randomly-generated strings are harder to work with than readable or empty strings. One participant referred to such strings as "garbage data". They went on to say that random strings are "harder when you are thinking of primary and foreign key [string columns], as you had to combine them, and there will be one letter difference, and it will be easier if it is real words". In particular, DOM-RND generates random string values, as shown in Figures 4.1(b) and 4.2(b). Comparing the similarity of values that have small differences requires more attention. One participant stated that small differences with characters are "trickier" when trying to review duplicates and references. After completing the survey, the experienced industry engineer said "the one I liked least is random values" (i.e., data generated by DOM-RND). Of the data generated, he stated "these are horrible, they are more distinct . . . but they do not mean anything. At least [readable strings], I can understand. But for this I had to compare each character".

Because they are both "more readable" and "pronounceable", participants also preferred non-random strings (e.g., those produced by DOM-COL, DOM-READ, and AVM-LM).

Concluding this KO, humans prefer readable, realistic strings to randomly-generated ones when understanding schema tests.

■ *It is Easy to Identify When* `NULL` *Violates* `NOT NULL` *Constraints* (KO5). `NULL` was confusing for participants when used with foreign keys and `CHECK` constraints, but as would be expected, their behaviour is straightforward to identify when used with `NOT NULL` constraints. Three participants made this comment. One participant stated after he finished the questions that "the `NOT NULL` constraints are the easiest to spot [violation of `NOT NULL`], followed by `PRIMARY KEY` constraints". Another participant commented on a test case that did not involve `NULL`: "nothing is `NULL`, so it is easy to see the ones [`INSERT` statements] that are `NULL` to see if they will be rejected".

To conclude this KO, it is clear that `NULL` has differing effects on test case comprehension, depending on the context in which it appears. When used with `NOT NULL` constraints, human testers thought that the behaviour of a test was obvious.

■ *Empty Strings Look Strange* (KO6)*, But They Are Helpful* (KO7). The AVM-D technique uses empty strings as the initial value for string columns in `INSERT` statements, only modifying them as required by the goal of the test case, as illustrated by the examples in Figures 4.1(b) and 4.2(b).

When a question involving a test case generated by AVM-D was revealed to one of the participants, he said "this is difficult". However, he changed his mind afterward, saying that one could see the "differences and similarities between `INSERTs`", which helped him to identify parts of the `INSERT` statements that affected the behaviour of the overall test case.

Another participant stated that a test case with default values was "a good one" because "zeros are easy to read". However, when the same participant first encountered empty strings he said that they were "weird". The experienced industry engineer liked empty strings because "they are easy to skip over to get to the important data". Reflecting on test effectiveness, he also said "empty strings are boundary values that need to be tested".

To conclude this KO, empty strings help to denote unimportant data, an crucial cue in SQL test comprehension.

The answer to RQ2 includes many thought-provoking observations. Participants raised issues concerning the use of `NULL` (KOs 1, 2 and 5), suggesting its judicious use in test data generation. There were positive comments about default values (KO7), readable strings (KO4), and unenthusiastic comments about negative numbers (KO3) and random strings (KO4). In the subsequent discussion section these factors will be explored.

4.5 Discussion

There are several factors that influence the understanding of automatically generated SQL tests, as evident from the think aloud study. This section investigates the frequency of these factors in the test cases generated by each method, explaining whether they aid or hinder successful test comprehension.

Table 4.3: Test Case Factors by Technique, Including The Percentage That Silent Study Participants Correctly Comprehended

Technique	**`NULL`s**			**Negatives**			**Word Frequency**		**Distinct Values**			**Percentage Correct**
	Median	Mean	Total	Median	Mean	Total	Median	Mean	Median	Mean	Total	
AVM-D	0	0.3	5	0	0.1	2	4	5.1	4	3.8	68	84%
AVM-LM	1	1.1	21	1	1.4	25	1	1.2	10.5	12	216	72%
DOM-COL	1.5	1.4	26	0	0	0	1	1.2	10.5	11.3	203	74%
DOM-RND	2	1.5	27	0.5	1.1	19	1	1.1	10	11.3	203	61%
DOM-READ	2	2.7	39	1	1.4	26	1	1.2	10.5	11.3	204	72%

Frequency of `NULL`. Table 4.3 shows the median, mean, and total occurrences of `NULL` in the 18 test cases generated by each technique for the two schemas used in the study. Test cases generated by AVM-D did not have many `NULL`s (5 in total) compared to the other techniques, which involved 20 or more occurrences, with 39 for DOM-READ. AVM-D's tests had the highest comprehension rate in the silent study: 84% of questions involving them were answered correctly, as shown in the table. Conversely, test generation techniques leading to many occurrences of `NULL` (e.g., DOM-RND and DOM-READ) had the lowest comprehension rates. The TAS revealed that participants got confused with `NULL`s on columns involving integrity constraints, but which did not also have `NOT NULL` constraints defined on them. This suggests two strategies: (1) generate `NULL`s in these scenarios, helping testers to understand the behaviour of `NULL` in schemas and test edge cases that detect more faults, as reported in the previous chapter or (2) limit the use of `NULL` in order to expedite the human oracle process.

Negative Numbers. Table 4.3 shows the median, mean, and total occurrences of negative numbers in test cases generated by each technique. DOM-COL generates numeric values through the use of sequential integers, and therefore did not produce test cases with any negative numbers. AVM-D's test cases only contained two occurrences of negative numbers, while other techniques involved 20 or more occurrences. AVM-D and DOM-COL were two of the best performers in terms of test case comprehension for RQ1, but there is not a significant difference in the number of questions that participants correctly answered between DOM-COL and the other techniques. Therefore, the data gives weak evidence that negative numbers affect test case comprehension; however, negative numbers are important to test boundaries, and as such the decision to include them needs to balance thoroughness of the testing process with human comprehension of test cases.

Repetitious Values. TAS participants commented that the AVM-D's use of many empty strings helped them to identify the important parts of the test case. Critically, the smaller the number of distinct values in a test case, the smaller the amount of information the human had to understand. Table 4.3 shows that AVM-D involved the smallest number of distinct values in the test cases generated (e.g., 68), while the number of distinct values for the other techniques was more or less similar (e.g., approximately 200). The frequency of string values follows an inverse pattern. AVM-D's test cases received the highest percentage of correctly comprehended test cases. Once again, the results suggest that repetitious values are a positive factor for the database schema tests. Moreover, unlike the other factors (i.e., the use of `NULL` or negative numbers), repeating values or using suitable defaults (e.g., empty strings or zero values) for unimportant aspects of a test case may not limit a human tester's ability to understand a schema test's behaviour.

Default values showed to be beneficial, especially empty strings and zeros in this study. This was because empty strings helped participants to skip to the important data and remove the task of comparing characters which can be a tedious and error-prone task.

Table 4.3 shows that AVM-D have a low distinct values, which means it has very low unique values compared to other techniques. Such low distinct values can help testers to review important data and skip values that are not related to the test requirements. Hence, such low distinct values helped participants to see similarities between different `INSERT` statements, which made it easier to identify issues. This is why users also got high scores when encountering default values. However, empty strings where reported to be "weird" by the think aloud experiment but that did not discourage the participants to get higher scores.

Table 4.3 shows the word frequency that do not including empty strings, that are generated by the techniques. We have not included empty strings as we want to review readable values. The mean word frequency shows that DOM-RND has the lowest word frequency and AVM-D has the highest. Which means that DOM-RND has high number of unique strings generated per test case than AVM-D, that is very low distinct values. Which mean it harder to spot differences between string values and the think aloud participants reported that they had harder time comparing characters between string values that are random. This results shows equally generated values (i.e. default values) has higher chance to be read easily than unique values that might confuses the tester.

Readable Values. I developed three techniques to generate non-random, human-readable strings (e.g., AVM-LM, DOM-COL, and DOM-READ). The results for RQ1 do not suggest that this was the most important factor in test case comprehension. While these techniques did not produce test cases with the highest comprehension rate, they were also not the worst. In the TAS of RQ2, participants agreed that random strings were hard to understand, and therefore preferred readable strings. The experienced industry engineer was asked about the different types of strings produced by AVM-LM, DOM-COL, and DOM-READ. He said the following of DOM-READ: "...easy to compare them because I can read them. [I see] distinct values, but I prefer nouns and adverbs"; of strings generated by the AVM-LM: "nice because they are pronounceable"; of strings generated by DOM-COL: "[values are] easy to correlate" with column names. However, he also stated that DOM-COL should have "visually different words" to help distinguish between different values. Overall, while human-readable values seem helpful, the results suggest that they are not critical to SQL test case comprehension.

The responses to RQ1 and RQ2 and the results in Table 4.3 highlight the factors that influence human comprehension of schema tests. The results suggest that the frequency of `NULL`s, existence of negative numbers, repetition of data values, and presence of readable values can influence the understandability of automatically generated tests. This means that both manual testers and the creators of automated testing tools should consider these issues as they may influence whether humans can understand tests and effectively complete testing tasks.

4.6 Summary

This chapter presented a study of the factors that make SQL test cases understandable for human testers and revealed the followings:

1. `NULL` is confusing for testers. Testers find the behaviour of `NULL` difficult to predict when used in conjunction with different integrity constraints such as foreign keys and `CHECK` constraints, suggesting the need for their judicious use.

2. Negative numbers require testers to think harder. Testers prefer positive numeric values, although, from a testing perspective, negative numbers should not be avoided altogether.

3. Simple repetitions for unimportant test values help testers. If only the important data values in the test case vary, while all others are held constant, a tester can easily focus on the non-trivial aspects of a test case to understand its behaviour.

4. Readable string values. Testers prefer to work with human-readable strings rather than randomly generated strings.

Therefore, this chapter evaluated and identified factors to help lower the qualitative human oracle cost. However, test data generators can generate many tests depending on the coverage criteria and the number of integrity constraints with a schema. Also, many test cases within a test suite can take long time to run when schemas change. Therefore, generated test suite can be reduced using reduction techniques to help both the human tester, lowering the quantitative human oracle cost, and help with regression testing (i.e., old tests ran on changed code). Hence, the following chapter will empirically evaluate reduction techniques, that are general purpose, in the context of database schema testing. Also, creating a new technique that reduce test suite by discarding and merging redundant test cases.

Chapter 5

STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases

The content of this chapter is based on the submitted and accepted work during this PhD, which will be published and presented in the International Conference on Software Testing, Verification and Validation (ICST) 2020 [161].

5.1 Introduction

The previous chapter investigated understandability factors associated with automagically generated test cases and recommended minimising the random values, use of `NULL`s and negative numbers, while repeating values in `INSERT` statements. These recommendations will help testers inspect generated tests and easily act as an oracle. However, automated techniques can generate many tests that are required to be inspected and executed to identify faults. This will increase testers inspection efforts and waste time awaiting results. Therefore, this chapter will investigate reduction techniques for database schema test suites.

Many real-world database applications contain complex and rapidly evolving database schemas [35, 22, 133], suggesting the need for efficient approaches to regression testing. Since these schemas contain many tables, columns, and integrity constraints, state-of-the-art automated test data generators for schemas can generate numerous tests that cover many test requirements [16, 17]. One approach to the regression testing of a database schema involves re-running the automatically generated tests after a schema modification, with follow-on steps that ensure the test suite's continued effectiveness by adequacy assessment through both coverage and mutation analysis [136]. The prohibitive cost of repeated test suite execution and test adequacy assessment suggests the need for test suite reduction methods that can minimise the test suite to those tests that are essential for maintaining the schema's correctness during its evolution.

Automatically generated schema tests construct complex database states that are often intertwined, thereby leading to test dependencies that are not explicitly captured by the requirement that the test was designed to cover. Since traditional test suite reduction methods (e.g., [25, 26, 27]) discard tests only when they cover the same requirements, they are not well-suited to reducing test suites for database schemas. Since Section 5.6's results show that these traditional reducers overlook up to 539 opportunities for test data merging in a complex schema like *iTrust*. Thus, this chapter presents a novel approach to test suite reduction, called **S**chema **T**est **I**ntegrity **C**onstraints **C**ombination for **E**fficient **R**eduction (STICCER), that discards tests that redundantly cover requirements while also merging those tests that produce similar database states. STICCER creates a reduced test suite, thus decreasing both the number of database interactions and restarts and lessening the time needed for test suite execution and mutation analysis.

Using the same previous 34 relational database schemas (in Chapter 3) and test data generated by two state-of-the-art methods, we experimentally compared STICCER to two greedy test suite reduction techniques and a random method. The results show that reduced test suites produced by STICCER are up to 5X faster than the original test suite and 2.5X faster than reduced suites created by the traditional methods, often leading to significant decreases in mutation analysis time. STICCER's tests always preserve the coverage of the test suite and rarely lead to a drop in the mutation score, with a maximum decrease in fault detection of 3.2%. In summary, this chapter's contributions are as follows:

1. A novel test suite reduction method, called STICCER, that quickly and effectively reduces database test suites by discarding and merging redundant schema tests.
2. Traditional test suite reduction methods implemented in the context of database schema testing.
3. An empirical study of STICCER's effectiveness when it reduces test suites from two state-of-the-art test data generators, compared to two traditional reducers and a random baseline, and using two schemas. The results highlight: (i) the limitations of existing methods, (ii) the decrease in tests and database interactions, (iii) the impact on the mutation score, and (iv) the change in the time taken for test execution and mutation analysis.

To support the replication of this chapter's experimental results and to facilitate the testing of relational database schemas, the proposed techniques are implemented into *SchemaAnalyst* [145] and the procedures into scripts. Replication instructions are available in Appendix B.3.

5.2 Motivation

Following the same consistency of the previous chapter, Figure 5.1 shows examples of test cases for the *BrowserCookies* and generated by AVM-D (part b(i)) and DOM-RND (part b(ii)). The test requirement for the AVM-D test case is to satisfy the `UNIQUE` constraint of the schema involving `name`, `host` and `path`, while the goal of DOM-RND's requirement is to violate it. For each test, assuming an initially empty `cookies` table, "setup" `INSERT` statements (*-S_1 and *-S_2) are needed to put the database into the required state for testing the constraint — since uniqueness cannot be tested unless there is already some data in the database for comparison purposes.

In both cases, the S_2-suffixed statement inserts data for the `cookies` table, but since it has a foreign key to `places`, a prior `INSERT` (*-S_1) must first be

(i) AVM-D generated test case to satisfy the compound `UNIQUE` key

$A\text{-}S_1$	`INSERT INTO places(host, path, title, visit_count, fav_icon_url)` `VALUES ('', '', '', 0, '')`
$A\text{-}S_2$	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)` `VALUES (0, '', '', 0, 0, 0, '', '')`
$A\text{-}A_1$	`INSERT INTO places(host, path, title,visit_count, fav_icon_url)` `VALUES ('a', '', '', 0, '')`
$A\text{-}A_2$	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)` `VALUES (1, '', '', 0, 0, 0, 'a', '')`

(ii) DOM-RND generated test case to violate the compound `UNIQUE` key

$D\text{-}S_1$	`INSERT INTO places(host, path, title, visit_count, fav_icon_url)` `VALUES ('aqrd', 'xj', 'vnobtpvl', 0, 'dmnofpe')`
$D\text{-}S_2$	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)` `VALUES (0, 'ddfvkxnjg', '', 0, -801, -890, 'aqrd', 'xj')`
$D\text{-}A_1$	`INSERT INTO places(host, path, title,visit_count, fav_icon_url)` `VALUES ('vkjdkfc', 'xxfp', 'tp', -640, 'mdewsfaw')`
$D\text{-}A_2$	`INSERT INTO cookies(id, name, value, expiry, last_accessed, creation_time, host, path)` `VALUES (261, 'ddfvkxnjg', 'euer', NULL, NULL, NULL, 'aqrd', 'xj')`

Figure 5.1: Test case examples for the *BrowserCookies* relational database schema in Figure 4.1.

made to that table. This is so that the test does not fail before the `UNIQUE` constraint can be tested, as violation of the foreign key is not the focus of these particular tests. Following these "setup" `INSERT`s are statements referred to as the "action" `INSERT`s, since they perform the actual test (*-A_2) — or support it through ensuring that the foreign key relationship to the `places` table can be maintained (*-A_1). A-A_2 inserts a different combination of values for `name`, `host`, and `path` to A-S_2, ensuring that the integrity constraint is satisfied, while D-A_2 inserts the *same* values as D-S_2, so that the constraint is ultimately violated.

The coverage criteria for testing integrity constraints necessitate the coverage of many test requirements. Especially, the combination of coverage criteria, such as the "Clause-Based Active Integrity Constraint Coverage" (ClauseAICC), "Active Unique Column Coverage" (AUCC), and "Active Null Column Coverage" (ANCC) that are they strongest in finding faults. Although *SchemaAnalyst* automates the process of generating test cases to satisfy those test requirements, the resultant test suites can still be lengthy, further manipulating database state in an intertwined fashion. As such, the combination of coverage criteria that the experiments in Section 5.6 use to generate the test suites. This chapter investigates ways to reduce the size of these test suites while maintaining their coverage.

5.3 Test Suite Reduction

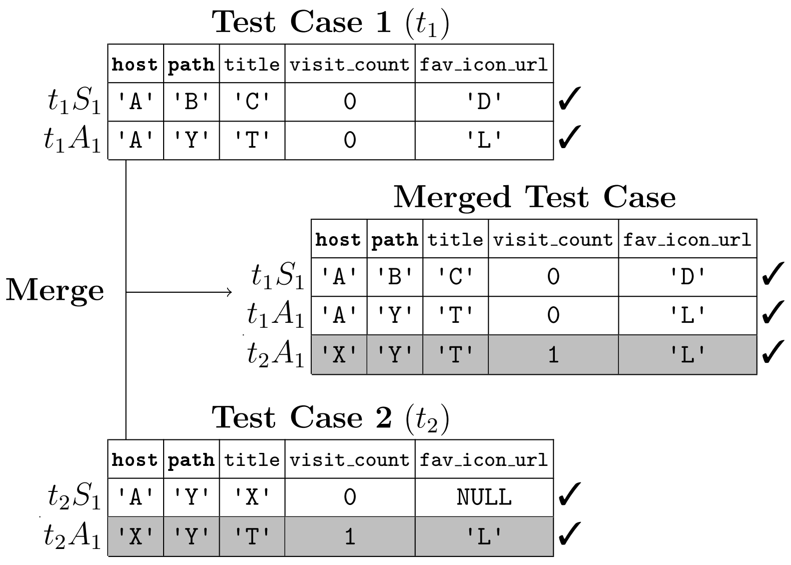
The task of producing a reduced test suite is equivalent to the minimal set cover problem, which is NP-complete [118]. However, several techniques are capable of effectively reducing test suite size in a way that is useful to developers. In chapter 2, three reduction techniques were reviewed and are implemented in *SchemaAnalyst* for this chapter result's section. The three reduction techniques are explained in Chapter 2 and they are as follows:

- A random reduction technique that randomly select test cases until all test requirements are covered.
- A greedy technique that reduced test suites by selecting test cases that cover most test requirements.
- The HGS technique that was developed by Harrold, Gupta, and Soffa [25] and uses set cardinality to reduce test suites.

5.4 The STICCER Approach

The more fault-finding and powerful a coverage criteria is, the more test requirements it involves, and the test suites needed to satisfy all of the those test requirements become larger as a result. For example, the *BrowserCookies* schema in Figure 4.1a has ten integrity constraints. A basic coverage criterion that simply satisfies and violates each constraint would therefore have 20 test requirements, The more complex combination of ClauseAICC, ANCC and AUCC, with higher fault revealing power [7], has 71. Although *SchemaAnalyst* automates the generation of test cases, the test suites can become large as a result of the number of test requirements must satisfy: *SchemaAnalyst* treats each test requirement as a separate "target" for test case generation, unless the test requirement is a duplicate or subsumed by some other.

As explained in the last section, and as shown by Figure 5.1, each test case involves a setup "cost", i.e., `INSERT` statements that are needed to get the database into some state required for the test requirement. For instance, in Figure 5.1, the database needs to have some data in it for the `UNIQUE` constraint to be properly tested, else there will be nothing in the database to test the "uniqueness" of the inserted data that forms the last `INSERT` statement of the test. Since the table of interest involves a foreign key, data needs to be into the referenced table as well, otherwise the test will fail for reasons other than the `UNIQUE` constraint that it is supposed to test. Furthermore, the database state must be reset following each test so that it does not "pollute" any following tests and introduce unintended behaviour and/or flakiness that might compromise testing [162].



Test Case 1 (t_1)

	host	path	title	visit_count	fav_icon_url	
t_1S_1	'A'	'B'	'C'	0	'D'	✓
t_1A_1	'A'	'Y'	'T'	0	'L'	✓

Merged Test Case

	host	path	title	visit_count	fav_icon_url	
t_1S_1	'A'	'B'	'C'	0	'D'	✓
t_1A_1	'A'	'Y'	'T'	0	'L'	✓
t_2A_1	'X'	'Y'	'T'	1	'L'	✓

Test Case 2 (t_2)

	host	path	title	visit_count	fav_icon_url	
t_2S_1	'A'	'Y'	'X'	0	NULL	✓
t_2A_1	'X'	'Y'	'T'	1	'L'	✓

Figure 5.2: An example STICCER merging of two test cases with the same test coverage behaviour. The test cases involve the `places` table of the *BrowserCookies* schema. The tick marks indicate the data is successfully admitted into the database.

The *first observation* was that integrity constraint tests often share common sequences of setup `INSERT` statements that could be shared across different test cases to reduce setup/teardown time when running the test suite, resulting in fewer overall `INSERT` statements for a human to understand when maintaining the test suite. For example, Figure 5.2 shows two test cases, t_1 and t_2, designed to test the inclusion of the two columns of a compound primary key belonging to the `places` table of the *BrowserCookies* schema in Figure 5.1. The first, t_1, tests the uniqueness of the `host` column while t_2 tests the uniqueness of the `path` column. As the figure shows, the setup part of t_2 can be thrown away (statement t_2S_1), with the "action" part (statement t_2A_1) appended to the end of t_1. The new, merged test case covers both test requirements of the original two tests.

The *second observation* was that some `INSERT` statements pertaining to foreign keys in a test case are redundant and can be removed. Take the example of Figure 4.1b(ii) and the test case generated by DOM-RND. Here, the test requirement is to violate the `UNIQUE` constraint of the schema. This involves two `INSERT` statements to the `cookies` table (D-S_2 and D-A_2), with the second `INSERT` (D-A_2) replicating the data values for the columns involved in the constraint of the first (D-S_2), so that they clash with those already in the database. Because the `cookies` has a foreign key to the `places` table, the test case involves `INSERT` statements to that table also (D-S_1 and D-A_1) — one to support each `INSERT` to `cookies`. This is to ensure data is present to satisfy the foreign key relationship (since violating this constraint would mean that the test requirement for this test would not be fulfilled). Yet, in this particular case D-A_1 is redundant. Since the columns of the `UNIQUE` constraint are also involved in the foreign key, and both `INSERT` statements to `cookies` must have the same data for those columns, those `INSERT` statements both rely on D-S_1 to fulfil the foreign key relationship. Note that this is not always case, since the corresponding `INSERT` to the `places` table is needed for the AVM-D test case in part b(i), as the test requirement there is to satisfy the unique constraint, and as such D-S_2 and D-A_2 need to be distinct, which in turn means the foreign key values must also be distinct.

To handle both of these issues as part of an approach to reduce the size of relational database schema integrity constraint tests, a technique was implemented and called "STICCER", which stands for "**S**chema **T**est **I**ntegrity **C**onstraints **C**ombination for **E**fficient **R**eduction". STICCER builds on the standard greedy approach to test suite reduction by merging tests (or "sticking" them together) — thereby sharing setup statements (and the associated teardown costs following the test) — and by removing redundant `INSERT`

```
Input:
$OTS$, the set of tests in the original test suite;
$TR$, a set of test requirements;
$S: t \to R$, mapping tests $t \in OTS$ to the requirements $R$ they cover, $R \subseteq TR$
Output: $RTS$, a reduced test suite ($RTS \subseteq OTS$)
Step 1: Ensure test suite is free of redundant INSERT statements ;
1  $OTS \leftarrow$ removeRedundantInserts($OTS$);
Step 2: Do greedy reduction on the original test suite (OTS) ;
2  $RTS \leftarrow \varnothing$;
3  $RTS \leftarrow Greedy(OTS, S)$ # The algorithm presented in Fig 2.20b ;
Step 3: Iterate through the reduced test suite, merging test cases where feasible ;
4  for $t_1 \in RTS$ do
5      for $t_2 \in RTS$ do
6          if checkMerge($t_1, t_2$) then
7              $t_1 \leftarrow$ merge($t_1, t_2$);
8              $RTS \leftarrow RTS - \{t_2\}$;
9          end
10     end
11 end
```

Figure 5.3: Overview of the STICCER algorithm

statements.

The overall algorithm for STICCER, as is shown by Figure 5.3, works as follows: The first step removes redundant `INSERT` statements from the existing set of test cases, through a function called removeRedundantInserts. This function checks all `INSERT` statements made to foreign key tables, and ensures that the foreign keys are actually referenced by other `INSERT`s in the test. If they are not, those `INSERT` statements are redundant, and are cut out of the test case under consideration.

STICCER then performs a greedy reduction on the test suite as a second step, before moving into the test case merging stage. STICCER iterates through the test suite comparing each test t_1 with each other test t_2. STICCER then checks for a potential merge through a function called checkMerge. The primary role of checkMerge is to assess if the proposed merged test will cover the same test requirements as the original two tests. If the first test, $t1$, leaves the database state in such a way that the behaviour of the "action" `INSERT` statements (i.e., t_2A_1 in the example of Figure 5.2) will behave differently after merging (i.e., the merged test does not cover the same test requirements as $t1$ and $t2$ combined), checkMerge will reject the potential merge. STICCER continues iterating through the test suite, checking the remaining tests with the newly created test for further merge possibilities. To help ensure that STICCER does not produce overly-long, unwieldy tests that are difficult to understand and maintain CHECKMERGE will only merge tests if the following, further conditions are met:

1. *The coverage requirements involve the same database table.* CHECKMERGE will not merge two tests if they are designed to test integrity constraints belonging to different tables. Each test must focus on the integrity constraints of one table only.

2. *The tests are intended to both satisfy or both violate aspects of the schema.* To simplify the intentions of each test, and make it easier to maintain and understand, each merged test will only attempt to satisfy the integrity constraints of a particular schema table, or violate them. This way, the human tester/maintainer knows what type of behaviour to expect from the `INSERT` statements of each of the final tests — that is, whether the data in them is intended to be accepted by the DBMS, or whether they are all supposed to be rejected.

3. *The tests both involve database setup, or they both do not.* Some tests do not involve any database setup at all (e.g., `CHECK` constraints, where the predicate is only concerned with the current row of data, rather than the state of the database), and these tests should not be merged together with those that do.

If CHECKMERGE permits a merge of tests, a function called MERGE then actually performs the merge, removing the setup `INSERT` statements from t_2, and appending it to the end of t_1. This test may undergo further merges with other tests in the suite as they are considered in turn by STICCER.

STICCER is implemented into *SchemaAnalyst*, which has functionality to statically analyse what test requirements are covered by a test case (e.g., the predicates in Section 2.3.5), and utilised of this to check merged tests when implementing the presented technique.

5.5 Empirical Evaluation

This section evaluates STICCER by comparing it to other reduction techniques in an empirical study that seeks to answer the following three research questions:

RQ1: Reduction Effectiveness. How effective is STICCER at reducing the number of test cases and `INSERT`s within each test case, compared to other test suite reduction techniques, while preserving the test requirements of the original suite?

RQ2: Impact on Fault Finding Capability. How is the fault-finding capability of the test suites affected following the use of STICCER and other test suite reduction techniques?

RQ3: Impact on Test Suite and Mutation Analysis Runtime. How are the running times of the reduced test suites and subsequent mutation analysis affected by test suite reduction?

5.5.1 Methodology

To evaluate STICCER, the experiments are configured the same as in Chapter 3 using the same diverse set of 34 schemas. The test suites were generated using AVM-D and DOM-RND with the strongest coverage criterion combination (ClauseAICC, ANCC, and AUCC), as it has a strong fault-finding capability. Using the *SchemaAnalyst* framework, the test suites were generated with the well-known SQLite as the target DBMS. Furthermore, the experiments are repeated 30 times with 100,000 iterations for each test requirement as in Chapter 3.

To answer RQ1, STICCER will be compared at reducing the test suites generated by *SchemaAnalyst* with implementations of the Random, Greedy, and HGS methods (introduced in Section 2.4 and iterated in Section 5.3). To ensure fairness of comparison with STICCER, the implementations of these techniques also removed redundant `INSERT` statements from test suites (using the REMOVEREDUNDANTINSERTS function discussed in Section 5.5.1). I also calculated the effectiveness of reduction for the test suite size and number of `INSERT`s using Equation 5.1.

$$(1 - \frac{\textit{No. of test cases in the reduced test suite}}{\textit{No. of test cases in the original test suite}}) \times 100 \qquad (5.1)$$

The results will report the median values of this equation for each reduction method for the 30 test suites generated for each schema with the two test generation methods, and calculate statistical significance and effect sizes as detailed in the next section. Finally, reporting the number of merges STICCER was capable of to reduce test suites for the two different test generation techniques, AVM-D and DOM-RND.

To answer RQ2, the fault-finding capability of the reduced test suites will be investigated using mutation analysis. For this, the mutation analysis techniques implemented into *SchemaAnalyst* is used, which adopt Wright et al.'s [101] mutation operators for integrity constraints. These operators add, remove, and exchange columns in primary key, unique, and foreign key constraints, invert `NOT NULL` constraints, remove `CHECK` constraints and mutate their relational operators. The mutation score will be calculated for each reduced test suite, a percentage of mutants that are "killed" (i.e., detected) by the tests.

To answer RQ3, times needed by each reduction algorithm are tracked to reduce test suites, and the time needed to perform mutation analysis using the reduced suites.

All the experiments were performed on a Linux workstation running Ubuntu 14.04 with a 3.13.0–44 GNU/Linux 64-bit kernel, quad-core 2.4GHz CPU, and 12GB of RAM. Also, using SQLite version 3.8.2 with "in-memory" mode, following the prior experiments in Chapter 3.

5.5.2 Statistical Analysis

Because the techniques are stochastic, and because recording wall-clock timings for the experiment is potentially subject to interferences that cannot be controlled (e.g., operating system interrupts), thus experiments are repeated 30 times. As assumptions about the normality of the resulting distributions cannot be made, non-parametric statistical measures are applied, including the Mann-Whitney U-test and the $\hat{A}$ effect size metric of Vargha and Delaney [143] as recommended by Arcuri and Briand in their guide to statistics and software engineering experiments [142]. In the following tables of the results section, we reported if a technique was statistically better or worse to some other by formatting the statistics for those techniques in bold, using the "∇" symbol if the secondary technique was *significantly worse* and the "Δ" symbol if it was *significantly better*, and an asterisk if the effect size was large (i.e., $\hat{A} < 0.29$ or > 0.71), following the suggested cut-offs from Vargha [143].

5.5.3 Threats to Validity

External Validity. The empirical study used a diverse set of schemas taken from past studies [17, 7]. While these schemas support the claims made in the following sections, it is impossible to claim that they are representative of all schemas — yet obtaining such a suitably representative set is equally difficult. Nevertheless, the set of schemas used come from a variety of sources and have a range of size and complexity.

Internal Validity. The stochastic behaviour of the test data generators (and the possibility of results are obtained by chance and are thus unrepresentative) and the use of wall-clock timings, which are subject to interferences that are out of my control. To mitigate both these issues, the experiments were repeated 30 times. Also, following the advice of Arcuri and Briand [142] to mitigate errors in the statistical analysis of the results, for example by using non-parametric hypothesis tests.

Finally, threats arising from defects in the implementation of STICCER and the reduction techniques were controlled with checking the results on selected or all schemas, where appropriate. For example, the techniques studied in this chapter aim to reduce test suites while maintaining test coverage. Therefore, all reduction was achieved without lowering test coverage with respect to the original test suites concerned (and this was indeed the case).

Construct Validity. The measure of reduction effectiveness is based on the number of test cases and `INSERT`s following Yoo and Harman [119].

However, other measurement such as the test suite execution times is not reported but the mutation analysis timings showcase the efficiency gains of reduced test suites.

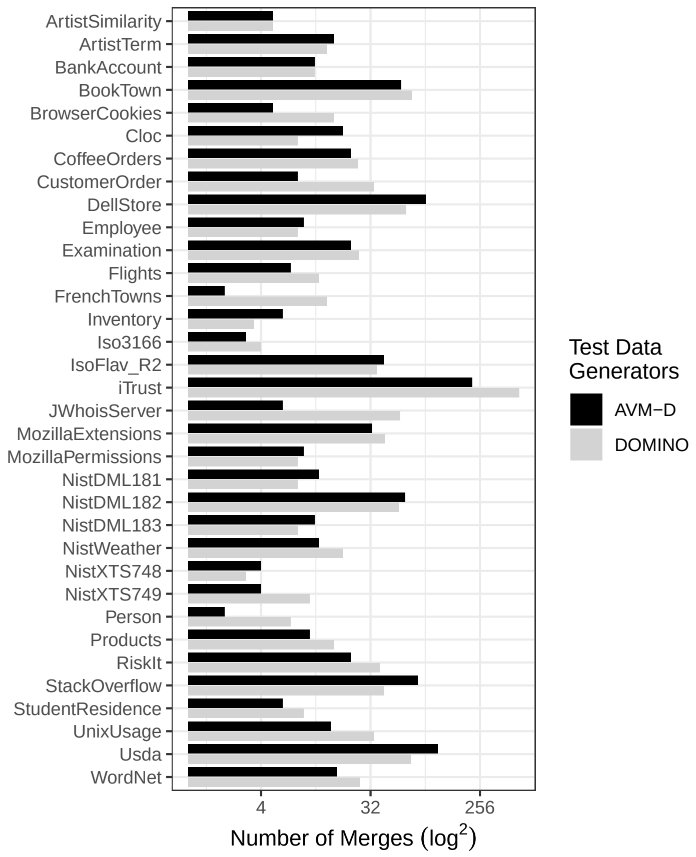


Figure 5.4: Median number of test case merges made by STICCER (log_2 scale)

Table 5.1: Median Reduction Effectiveness for Test Suites

Reduction effectiveness is calculated using the higher–is-better formula of Equation 5.1.

	AVM-D				DOM-RND			
Schema	STICCER	Random	Greedy	HGS	STICCER	Random	Greedy	HGS
ArtistSimilarity	63% (7/19)	*▼**32%** (13/19)	*▼**32%** (13/19)	*▼**42%** (11/19)	63% (7/19)	*▼**32%** (13/19)	*▼**37%** (12/19)	*▼**32%** (13/19)
ArtistTerm	65% (15/43)	*▼**28%** (31/43)	*▼**30%** (30/43)	*▼**37%** (27/43)	65% (15/43)	*▼**28%** (31/43)	*▼**30%** (30/43)	*▼**30%** (30/43)
BankAccount	68% (12/37)	*▼**35%** (24/37)	*▼**38%** (23/37)	*▼**43%** (21/37)	70% (11/37)	*▼**38%** (23/37)	*▼**43%** (21/37)	*▼**49%** (19/37)
BookTown	58% (113/269)	*▼**35%** (175/269)	*▼**38%** (167/269)	*▼**46%** (144/269)	68% (87/269)	*▼**37%** (168/269)	*▼**42%** (156/269)	*▼**49%** (138/269)
BrowserCookies	62% (27/71)	*▼**51%** (35/71)	*▼**56%** (31/71)	*▼**59%** (29/71)	76% (17/71)	*▼**46%** (38/71)	*▼**55%** (32/71)	*▼**56%** (31/71)
Cloc	90% (4/40)	*▼**43%** (23/40)	*▼**48%** (21/40)	*▼**50%** (20/40)	85% (6/40)	*▼**48%** (21/40)	*▼**51%** (20/40)	*▼**57%** (17/40)
CoffeeOrders	64% (32/90)	*▼**37%** (57/90)	*▼**41%** (53/90)	*▼**42%** (52/90)	74% (23/90)	*▼**39%** (55/90)	*▼**44%** (50/90)	*▼**47%** (48/90)
CustomerOrder	40% (76/126)	*▼**32%** (86/126)	*▼**33%** (84/126)	*▼**37%** (79/126)	63% (46/126)	*▼**33%** (85/126)	*▼**35%** (82/126)	*▼**39%** (77/126)
DellStore	86% (24/177)	*▼**33%** (118/177)	*▼**35%** (115/177)	*▼**38%** (110/177)	71% (52/177)	*▼**36%** (113/177)	*▼**41%** (105/177)	*▼**42%** (102/177)
Employee	74% (10/38)	*▼**46%** (20/38)	*▼**50%** (19/38)	*▼**50%** (19/38)	80% (8/38)	*▼**53%** (18/38)	*▼**61%** (15/38)	*▼**61%** (15/38)
Examination	72% (30/107)	*▼**50%** (54/107)	*▼**51%** (52/107)	*▼**52%** (51/107)	87% (14/107)	*▼**57%** (46/107)	*▼**64%** (38/107)	*▼**64%** (38/107)
Flights	69% (19/62)	*▼**49%** (32/62)	*▼**58%** (26/62)	*▼**58%** (26/62)	71% (18/62)	*▼**45%** (34/62)	*▼**52%** (30/62)	*▼**53%** (29/62)
FrenchTowns	40% (32/53)	*▼**32%** (36/53)	*▼**36%** (34/53)	*▼**34%** (35/53)	60% (21/53)	*▼**32%** (36/53)	*▼**32%** (36/53)	*▼**34%** (35/53)
Inventory	67% (6/18)	*▼**33%** (12/18)	*▼**39%** (11/18)	*▼**44%** (10/18)	78% (4/18)	*▼**44%** (10/18)	*▼**56%** (8/18)	*▼**56%** (8/18)
IsoFlav_R2	75% (45/177)	*▼**45%** (96/177)	*▼**49%** (90/177)	*▼**50%** (88/177)	81% (34/177)	*▼**52%** (85/177)	*▼**60%** (70/177)	*▼**62%** (66/177)
Iso3166	58% (5/12)	*▼**25%** (9/12)	*▼**33%** (8/12)	*▼**33%** (8/12)	58% (5/12)	*▼**29%** (8/12)	*▼**33%** (8/12)	*▼**42%** (7/12)
iTrust	57% (646/1517)	*▼**38%** (934/1517)	*▼**43%** (872/1517)	*▼**44%** (847/1517)	85% (235/1517)	*▼**44%** (849/1517)	*▼**49%** (776/1517)	*▼**50%** (754/1517)
JWhoisServer	37% (100/158)	*▼**31%** (109/158)	*▼**33%** (106/158)	*▼**35%** (103/158)	70% (48/158)	*▼**32%** (107/158)	*▼**35%** (103/158)	*▼**37%** (99/158)
MozillaExtensions	75% (57/229)	*▼**51%** (112/229)	*▼**60%** (92/229)	*▼**50%** (115/229)	85% (35/229)	*▼**55%** (102/229)	*▼**63%** (84/229)	*▼**64%** (83/229)
MozillaPermissions	73% (9/33)	*▼**42%** (19/33)	*▼**48%** (17/33)	*▼**48%** (17/33)	88% (4/33)	*▼**55%** (15/33)	*▼**58%** (14/33)	*▼**64%** (12/33)
NistDML181	79% (8/38)	*▼**47%** (20/38)	*▼**53%** (18/38)	*▼**53%** (18/38)	76% (9/38)	*▼**45%** (21/38)	*▼**55%** (17/38)	*▼**58%** (16/38)
NistDML182	89% (20/190)	*▼**53%** (90/190)	*▼**57%** (82/190)	*▼**57%** (81/190)	89% (21/190)	*▼**52%** (90/190)	*▼**60%** (76/190)	*▼**62%** (73/190)
NistDML183	82% (6/34)	*▼**44%** (19/34)	*▼**47%** (18/34)	*▼**53%** (16/34)	76% (8/34)	*▼**41%** (20/34)	*▼**50%** (17/34)	*▼**53%** (16/34)
NistWeather	64% (20/56)	*▼**39%** (34/56)	*▼**45%** (31/56)	*▼**43%** (32/56)	82% (10/56)	*▼**39%** (34/56)	*▼**45%** (31/56)	*▼**46%** (30/56)
NistXTS748	62% (6/16)	*▼**34%** (10/16)	*▼**38%** (10/16)	*▼**44%** (9/16)	69% (5/16)	*▼**41%** (10/16)	*▼**50%** (8/16)	*▼**50%** (8/16)
NistXTS749	57% (15/35)	*▼**43%** (20/35)	*▼**46%** (19/35)	*▼**51%** (17/35)	69% (11/35)	*▼**40%** (21/35)	*▼**49%** (18/35)	*▼**49%** (18/35)
Person	50% (10/20)	*▼**30%** (14/20)	*▼**40%** (12/20)	*▼**40%** (12/20)	80% (4/20)	*▼**30%** (14/20)	*▼**30%** (14/20)	*▼**35%** (13/20)
Products	67% (17/52)	*▼**40%** (31/52)	*▼**44%** (29/52)	*▼**48%** (27/52)	69% (16/52)	*▼**38%** (32/52)	*▼**44%** (29/52)	*▼**46%** (28/52)
RiskIt	51% (122/250)	*▼**41%** (148/250)	*▼**43%** (142/250)	*▼**48%** (130/250)	63% (92/250)	*▼**44%** (140/250)	*▼**48%** (129/250)	*▼**52%** (120/250)
StackOverflow	93% (12/171)	*▼**46%** (92/171)	*▼**48%** (89/171)	*▼**50%** (86/171)	84% (28/171)	*▼**53%** (80/171)	*▼**60%** (68/171)	*▼**60%** (68/171)
StudentResidence	62% (13/34)	*▼**38%** (21/34)	*▼**41%** (20/34)	*▼**44%** (19/34)	74% (9/34)	*▼**41%** (20/34)	*▼**47%** (18/34)	*▼**47%** (18/34)
UnixUsage	52% (70/147)	*▼**41%** (86/147)	*▼**43%** (84/147)	*▼**47%** (78/147)	73% (40/147)	*▼**44%** (83/147)	*▼**48%** (76/147)	*▼**50%** (73/147)
Usda	88% (30/247)	*▼**39%** (152/247)	*▼**40%** (147/247)	*▼**44%** (139/247)	78% (54/247)	*▼**44%** (139/247)	*▼**49%** (125/247)	*▼**51%** (121/247)
WordNet	58% (50/118)	*▼**36%** (76/118)	*▼**42%** (69/118)	*▼**44%** (66/118)	64% (43/118)	*▼**35%** (77/118)	*▼**40%** (71/118)	*▼**44%** (66/118)
Minimum	37%	25%	30%	33%	58%	28%	30%	30%
Average	66%	39%	43%	46%	74%	42%	48%	50%
Maximum	93%	53%	60%	59%	89%	57%	64%	64%

Table 5.2: Median Reduction Effectiveness for SQL INSERT statements (Database Interactions) in Reduced Test Suites

Reduction effectiveness is calculated using the higher–is-better formula of Equation 5.1.

	AVM-D								DOM-RND							
Schema	STICCER		Random		Greedy		HGS		STICCER		Random		Greedy		HGS	
ArtistSimilarity	48%	(23/44)	*▼**34**%	(29/44)	*▼**34**%	(29/44)	*▼**45**%	(24/44)	50%	(22/44)	*▼**32**%	(30/44)	*▼**39**%	(27/44)	*▼**35**%	(28/44)
ArtistTerm	56%	(54/124)	*▼**34**%	(82/124)	*▼**36**%	(79/124)	*▼**45**%	(68/124)	56%	(54/124)	*▼**33**%	(84/124)	*▼**36**%	(79/124)	*▼**36**%	(79/124)
BankAccount	56%	(35/80)	*▼**36**%	(52/80)	*▼**38**%	(50/80)	*▼**44**%	(45/80)	57%	(34/80)	*▼**39**%	(49/80)	*▼**44**%	(44/80)	*▼**50**%	(40/80)
BookTown	44%	(225/403)	*▼**35**%	(262/403)	*▼**38**%	(250/403)	*△**47**%	(215/403)	49%	(206/403)	*▼**37**%	(254/403)	*▼**42**%	(233/403)	*▼**48**%	(208/403)
BrowserCookies	64%	(63/175)	*▼**56**%	(77/175)	*▼**62**%	(66/175)	*▼**64**%	(63/175)	69%	(55/175)	*▼**50**%	(87/175)	*▼**59**%	(71/175)	*▼**60**%	(70/175)
Cloc	62%	(23/60)	*▼**42**%	(35/60)	*▼**48**%	(31/60)	*▼**50**%	(30/60)	61%	(24/60)	*▼**45**%	(33/60)	*▼**48**%	(31/60)	*▼**55**%	(27/60)
CoffeeOrders	61%	(107/273)	*▼**41**%	(162/273)	*▼**45**%	(149/273)	*▼**48**%	(143/273)	65%	(96/273)	*▼**43**%	(156/273)	*▼**49**%	(139/273)	*▼**51**%	(132/273)
CustomerOrder	39%	(290/475)	*▼**33**%	(317/475)	*▼**36**%	(305/475)	*△**39**%	(288/475)	56%	(208/475)	*▼**35**%	(310/475)	*▼**36**%	(302/475)	*▼**41**%	(279/475)
DellStore	56%	(123/281)	*▼**37**%	(177/281)	*▼**39**%	(172/281)	*▼**42**%	(162/281)	50%	(141/281)	*▼**40**%	(169/281)	*▼**44**%	(156/281)	*▼**47**%	(150/281)
Employee	60%	(21/53)	*▼**42**%	(30/53)	*▼**47**%	(28/53)	*▼**47**%	(28/53)	62%	(20/53)	*▼**49**%	(27/53)	*▼**57**%	(23/53)	*▼**57**%	(23/53)
Examination	66%	(78/229)	*▼**49**%	(118/229)	*▼**50**%	(114/229)	*▼**52**%	(111/229)	74%	(60/229)	*▼**55**%	(102/229)	*▼**63**%	(86/229)	*▼**64**%	(83/229)
Flights	70%	(41/137)	*▼**57**%	(59/137)	*▼**66**%	(46/137)	*▼**66**%	(46/137)	58%	(58/137)	*▼**47**%	(73/137)	*▼**55**%	(62/137)	*▼**55**%	(62/137)
FrenchTowns	50%	(81/161)	*▼**41**%	(95/161)	*▼**47**%	(86/161)	*▼**43**%	(91/161)	52%	(77/161)	*▼**40**%	(97/161)	*▼**40**%	(96/161)	*▼**43**%	(92/161)
Inventory	54%	(13/28)	*▼**32**%	(19/28)	*▼**39**%	(17/28)	*▼**43**%	(16/28)	57%	(12/28)	*▼**43**%	(16/28)	*▼**54**%	(13/28)	54%	(13/28)
IsoFlav_R2	62%	(104/274)	*▼**46**%	(148/274)	*▼**50**%	(136/274)	*▼**50**%	(136/274)	64%	(100/274)	*▼**49**%	(138/274)	*▼**58**%	(116/274)	*▼**59**%	(111/274)
Iso3166	47%	(10/19)	*▼**26**%	(14/19)	*▼**37**%	(12/19)	*▼**37**%	(12/19)	42%	(11/19)	*▼**32**%	(13/19)	*▼**37**%	(12/19)	47%	(10/19)
iTrust	56%	(978/2204)	*▼**43**%	(1261/2204)	*▼**48**%	(1142/2204)	*▼**50**%	(1103/2204)	57%	(940/2204)	*▼**45**%	(1212/2204)	*▼**50**%	(1101/2204)	*▼**52**%	(1064/2204)
JWhoisServer	38%	(158/256)	*▼**36**%	(163/256)	38%	(158/256)	*△**40**%	(153/256)	47%	(136/256)	*▼**38**%	(159/256)	*▼**41**%	(152/256)	*▼**43**%	(145/256)
MozillaExtensions	71%	(105/356)	*▼**53**%	(168/356)	*▼**63**%	(130/356)	*▼**50**%	(179/356)	69%	(110/356)	*▼**52**%	(170/356)	*▼**60**%	(144/356)	*▼**61**%	(140/356)
MozillaPermissions	62%	(19/50)	*▼**42**%	(29/50)	*▼**48**%	(26/50)	*▼**48**%	(26/50)	66%	(17/50)	*▼**51**%	(24/50)	*▼**54**%	(23/50)	*▼**60**%	(20/50)
NistDML181	68%	(25/78)	*▼**50**%	(39/78)	*▼**55**%	(35/78)	*▼**53**%	(37/78)	65%	(28/78)	*▼**45**%	(43/78)	*▼**54**%	(36/78)	*▼**56**%	(34/78)
NistDML182	73%	(102/384)	*▼**56**%	(168/384)	*▼**61**%	(151/384)	*▼**61**%	(150/384)	74%	(98/384)	*▼**53**%	(180/384)	*▼**61**%	(149/384)	*▼**63**%	(144/384)
NistDML183	65%	(24/68)	*▼**46**%	(37/68)	*▼**46**%	(37/68)	*▼**53**%	(32/68)	62%	(26/68)	*▼**41**%	(40/68)	*▼**50**%	(34/68)	*▼**51**%	(33/68)
NistWeather	61%	(47/120)	*▼**45**%	(66/120)	*▼**52**%	(58/120)	*▼**49**%	(61/120)	63%	(44/120)	*▼**45**%	(66/120)	*▼**51**%	(58/120)	*▼**52**%	(58/120)
NistXTS748	48%	(12/23)	*▼**28**%	(16/23)	*▼**35**%	(15/23)	*▼**39**%	(14/23)	52%	(11/23)	*▼**37**%	(14/23)	*▼**48**%	(12/23)	*▼**48**%	(12/23)
NistXTS749	53%	(34/73)	*▼**45**%	(40/73)	*▼**47**%	(39/73)	53%	(34/73)	58%	(30/73)	*▼**41**%	(43/73)	*▼**49**%	(37/73)	*▼**49**%	(37/73)
Person	53%	(14/30)	*▼**37**%	(19/30)	*▼**50**%	(15/30)	*▼**50**%	(15/30)	47%	(16/30)	*▼**37**%	(19/30)	*▼**37**%	(19/30)	*▼**43**%	(17/30)
Products	65%	(50/144)	*▼**47**%	(76/144)	*▼**53**%	(68/144)	*▼**56**%	(63/144)	62%	(54/144)	*▼**46**%	(78/144)	*▼**51**%	(70/144)	*▼**53**%	(68/144)
RiskIt	50%	(342/687)	*▼**44**%	(384/687)	*▼**47**%	(366/687)	*△**51**%	(336/687)	54%	(318/687)	*▼**46**%	(373/687)	*▼**50**%	(346/687)	*▼**53**%	(322/687)
StackOverflow	64%	(93/257)	*▼**46**%	(139/257)	*▼**48**%	(134/257)	*▼**50**%	(129/257)	66%	(88/257)	*▼**51**%	(126/257)	*▼**57**%	(111/257)	*▼**57**%	(110/257)
StudentResidence	56%	(32/72)	*▼**39**%	(44/72)	*▼**42**%	(42/72)	*▼**46**%	(39/72)	60%	(29/72)	*▼**42**%	(42/72)	*▼**49**%	(37/72)	*▼**49**%	(37/72)
UnixUsage	58%	(252/595)	*▼**45**%	(328/595)	*▼**47**%	(313/595)	*▼**51**%	(293/595)	70%	(178/595)	*▼**47**%	(318/595)	*▼**52**%	(287/595)	*▼**54**%	(274/595)
Usda	59%	(157/381)	*▼**40**%	(227/381)	*▼**42**%	(220/381)	*▼**46**%	(206/381)	59%	(157/381)	*▼**44**%	(212/381)	*▼**50**%	(190/381)	*▼**52**%	(181/381)
WordNet	50%	(96/192)	*▼**40**%	(116/192)	*▼**47**%	(102/192)	*▼**49**%	(98/192)	49%	(97/192)	*▼**39**%	(117/192)	*▼**45**%	(106/192)	*▼**49**%	(98/192)
Minimum	38%		26%		34%		37%		42%		32%		36%		35%	
Average	57%		42%		46%		49%		59%		43%		49%		51%	
Maximum	73%		57%		66%		66%		74%		55%		63%		64%	

5.6 Experimental Results

5.6.1 RQ1: Reduction Effectiveness

Figure 5.4 shows the median number of test cases merges that STICCER was able to make with test cases generated by the AVM-D and DOM-RND. The highest figure is 539 merges for the *iTrust* schema, with test suites generated by DOM-RND. The *iTrust* schema is the largest that was studied (see Appendix A), with the highest number of integrity constraints (134). The schema with the next highest number of merges is *BookTown*, with 70 merges, for test suites generated by DOM-RND. *BookTown*'s original test suite size is 269. The smallest number of merges was 2 for *Person*, for test suites generated by AVM-D, which has an original test suite size of 20. Figure 5.4 shows the distribution of merge opportunities that were unavailable to the other, traditional reduction techniques we applied to database scheme test suites, which only remove test cases on the basis of overlapping coverage requirements.

Table 5.1 shows the median effectiveness of each of the reduction techniques at decreasing the number of test cases for each schema in each of the 30 test suites generated by AVM-D and DOM-RND respectively. Table 5.2 shows the effectiveness of each reduction technique at reducing the overall number of SQL statements (i.e., `INSERT`s) in those reduced suites. The "∇" symbol indicates that a technique's reduction score was significantly less than STICCER, while "Δ" indicates that it was significantly higher. The "$*$" symbol in both table denotes a large effect size for a technique when compared with STICCER. The numbers in brackets indicate the median number of test cases/`INSERT`s in reduced suites as a fraction of those in the original, unreduced test suite.

As the summary statistics show, STICCER was the most effective at reducing test suites, achieving up to 93% for *StackOverflow* schema, and a minimum 37% for *JWhoisServer* for test suites generated using AVM-D. For test suites generated by DOM-RND, STICCER achieved a maximum reduction of 89% with *NistDML182* and a minimum of 58% for *Iso3166*. It is worth noting that this minimum figure is greater than the maximum achievable with Random for DOM-RND produced test suites, and only 6% lower than the maximums achieved by Greedy and HGS, respectively. STICCER created reduced test suites that were statistically significantly smaller than the original test suites and those reduced by the other methods for all database schemas and with a large effect size.

Table 5.2 shows that the reduction in test cases achieved by STICCER was not the result of naively concatenating SQL statements from each constituent test case — STICCER also reduced the number of overall `INSERT` statements in the resulting test suites. As Table 5.2 shows, the average reduction in the number of `INSERT`s (i.e., the constituent statements making up each test case) that STICCER achieved was greater than any of the other three techniques studied.

On average, STICCER is more effective with test suites generated by DOM-RND than AVM-D, a fact also shown by Figure 5.4. This is because the default values used by AVM-D are repeated across `INSERT` statements, which makes them more difficult to merge them together across different test cases. The repeated values inadvertently trigger primary key and `UNIQUE` constraint violations when the same values appear for different `INSERT` statements from different test cases for particular columns. This results in a combined test case with different coverage requirements compared to its constituent originals — test cases that will be rejected by STICCER.

Comparing the average of the median reduction scores for each schema, HGS is the next most effective reduction technique following STICCER. STICCER achieves an average reduction of 66% and 74% for test suites generated by AVM-D and DOM-RND, respectively, while HGS achieves comparatively lower scores of 46% and 50%. HGS also performed worse overall with test suites generated by AVM-D compared to DOM-RND, but the differences we observed were not as marked as those with STICCER. It seems that DOM-RND is capable of producing test cases that cover more distinct sets of requirements than AVM-D, and with less of an intersect with other test cases, thereby making them more amenable for techniques based on removing redundant test cases to reduce.

Greedy was the third best performer, achieving marginally less reduction to overall test suite size than HGS. As expected, the baseline Random technique was the poorest performer.

In conclusion for RQ1, STICCER is the most effective at reducing the number of test cases and the overall number of `INSERT` statements in a test suite. Importantly, STICCER does this while preserving the coverage of the original test suite.

5.6.2 RQ2: Impact on Fault Finding Capability

Table 5.3 shows the median mutation scores for all the reduction and test generation techniques where a difference was recorded for one of the reduction techniques with respect to the original test suite (OTS). The "$\triangledown$" symbol in the table means that significance tests show that a technique's test suite obtained a significantly lower mutation score compared to the OTS. Perhaps surprisingly, differences were only observed for one of the reduction techniques for eight of the 34 schemas — for the rest, the same mutation score was recorded. For each of these schemas, differences were only experienced with test suites generated by AVM-D.

Table 5.3: Median Mutation Scores

Scores are expressed as percentage of mutants killed by the test suites concerned. "OTS" refers to the original, unreduced test suites.

	AVM-D					DOM-RND				
Schemas	OTS	STICCER	Random	Greedy	HGS	OTS	STICCER	Random	Greedy	HGS
BrowserCookies	86.5	▼**86.5**	86.5	▼**86.5**	▼**86.5**	96.6	96.6	96.6	96.6	96.6
FrenchTowns	83.3	*▼**80.3**	*▼**80.3**	*▼**80.3**	*▼**81.8**	95.5	95.5	95.5	95.5	95.5
iTrust	83.6	*▼**83.6**	*▼**83.6**	*▼**83.6**	*▼**83.6**	99.2	99.2	99.2	99.2	99.1
NistWeather	93.8	*▼**90.6**	93.8	*▼**90.6**	93.8	100.0	100.0	100.0	100.0	100.0
NistXTS749	92.0	92.0	▼**92.0**	92.0	*▼**88.0**	94.0	94.0	94.0	94.0	94.0
RiskIt	89.3	89.3	▼**89.3**	89.3	*▼**88.8**	99.5	99.5	99.5	99.5	99.5
UnixUsage	98.2	98.2	98.2	98.2	*▼**97.3**	100.0	100.0	100.0	100.0	100.0
WordNet	87.4	*▼**86.3**	▼**87.4**	*▼**86.3**	*▼**86.3**	99.0	99.0	99.0	99.0	99.0

For test suites generated by AVM-D, STICCER's reduced test suites had a mutation score significantly worse than the OTS for five schemas. Although statistically significant, the different does not register to the first decimal point for two schemas (*BrowserCookies* and *iTrust*) and the difference is at most 3.2% (for *NistWeather*). Random and Greedy were also significantly worse for five schemas with the AVM-D. Greedy performed almost identically to STICCER. Given that STICCER performs greedy reduction as one of its initial steps, this points to the loss of fault detection capability being down to test cases that were removed with duplicate due to coverage requirements, rather than test case merging. Finally, HGS was statistically significantly worse compared to the OTS for the highest number of schemas, namely eight in total. That is because HGS selects test cases that have the least diversity (e.g., more `INSERT`s with more default values) than Greedy.

For STICCER, the results show that merging of test cases does not affect the reduced test suite fault detection that was produced from the greedy technique. For STICCER and greedy, there are two schemas where test suites were originally generated by AVM-D that achieved lower scores after reduction — *FrenchTowns* and *NistWeather*. The differences are significant but not substantial. That is because the selected tests contain less distinct values that lower the capability of finding faults. For instance, changing column of a `FOREIGN KEY` will likely not fail tests as most values are equal and can be linked with parent tables. Or, changing a column of the compounded `UNIQUE` constraint will not be detected as values are equal and there will be no change of behaviour.

In all cases, DOM-RND generated suites are more robust to the reduction, likely because of diversity of test values that it generates. Conversely, the re-use of “default” values for AVM-D means that the loss of test cases and `INSERT`s through reduction results in a small loss of fault-finding capability.

In conclusion for RQ2, mutation scores of test suites were more or less preserved following reduction. While some test suites experienced a drop in mutation score, the difference was not substantial (3.2% maximum). Test suites generated by DOM-RND did not experience any loss of mutation score following the application of any of the reduction techniques.

Table 5.4: Median Mutation Analysis Times (Seconds) for STICCER versus the Original Test Suite

STICCER times are broken down into "RT" (time to reduce test suites), and "MT" (time for mutation analysis with STICCER), with the total compared to the original test suite (OTS) for a fair comparison.

	AVM-D				DOM-RND			
	OTS	STICCER			OTS	STICCER		
Schemas	Total	RT	MT	Total	Total	RT	MT	Total
ArtistSimilarity	0.09	0.12	0.05	*△**0.17**	0.10	0.13	0.05	*△**0.18**
ArtistTerm	0.54	0.14	0.22	*▼**0.37**	0.54	0.14	0.22	*▼**0.36**
BankAccount	0.45	0.17	0.19	*▼**0.36**	0.46	0.16	0.19	*▼**0.34**
BookTown	36.19	0.94	16.17	*▼**17.11**	36.50	1.07	10.85	*▼**11.92**
BrowserCookies	2.21	0.37	0.94	*▼**1.31**	2.34	0.37	0.73	*▼**1.09**
Cloc	0.27	0.15	0.07	*▼**0.22**	0.28	0.15	0.09	*▼**0.25**
CoffeeOrders	2.96	0.31	1.20	*▼**1.51**	2.98	0.29	1.05	*▼**1.34**
CustomerOrder	9.74	0.80	6.68	*▼**7.47**	9.78	0.84	4.56	*▼**5.40**
DellStore	8.04	1.42	1.93	*▼**3.35**	8.40	1.46	3.22	*▼**4.68**
Employee	0.34	0.20	0.13	▼**0.32**	0.36	0.18	0.12	*▼**0.30**
Examination	4.42	1.06	1.39	*▼**2.45**	4.48	1.11	1.00	*▼**2.12**
Flights	1.60	0.30	0.61	*▼**0.91**	1.68	0.41	0.68	*▼**1.09**
FrenchTowns	2.43	0.23	1.46	*▼**1.69**	2.44	0.23	1.23	*▼**1.46**
Inventory	0.10	0.12	0.05	*△**0.17**	0.10	0.12	0.04	*△**0.17**
IsoFlav_R2	9.80	0.70	2.83	*▼**3.52**	10.20	0.75	2.72	*▼**3.47**
Iso3166	0.04	0.11	0.02	*△**0.13**	0.05	0.11	0.03	*△**0.13**
iTrust	2297.86	150.17	959.31	*▼**1109.48**	2330.02	157.23	428.57	*▼**585.80**
JWhoisServer	8.91	1.64	5.62	*▼**7.25**	9.34	1.86	3.88	*▼**5.74**
MozillaExtensions	25.52	2.02	6.62	*▼**8.64**	26.61	2.38	5.39	*▼**7.78**
MozillaPermissions	0.23	0.15	0.08	0.23	0.24	0.15	0.07	*▼**0.22**
NistDML181	0.36	0.15	0.11	*▼**0.26**	0.38	0.15	0.12	*▼**0.28**
NistDML182	14.60	1.93	2.22	*▼**4.15**	14.99	2.00	2.43	*▼**4.43**
NistDML183	0.28	0.14	0.09	*▼**0.23**	0.29	0.14	0.10	*▼**0.25**
NistWeather	0.74	0.22	0.29	*▼**0.51**	0.76	0.23	0.25	*▼**0.48**
NistXTS748	0.08	0.12	0.04	*△**0.16**	0.08	0.11	0.04	*△**0.15**
NistXTS749	0.39	0.17	0.20	▼**0.37**	0.40	0.17	0.16	*▼**0.33**
Person	0.16	0.10	0.09	*△**0.19**	0.16	0.15	0.07	*△**0.22**
Products	1.04	0.14	0.42	*▼**0.57**	1.06	0.18	0.44	*▼**0.62**
RiskIt	44.40	1.41	23.15	*▼**24.57**	45.91	1.59	18.90	*▼**20.49**
StackOverflow	5.58	0.93	0.97	*▼**1.90**	5.76	1.16	1.42	*▼**2.58**
StudentResidence	0.43	0.16	0.20	*▼**0.36**	0.44	0.14	0.17	*▼**0.32**
UnixUsage	12.25	0.92	6.21	*▼**7.13**	12.34	0.95	3.83	*▼**4.79**
Usda	15.55	1.39	2.78	*▼**4.17**	16.44	1.52	4.15	*▼**5.68**
WordNet	3.89	0.37	1.86	*▼**2.23**	3.93	0.38	1.69	*▼**2.07**

5.6.3 RQ3: Impact on Test Suite and Mutation Analysis Runtime

Table 5.4 shows the median time for performing reduction with STICCER and mutation analysis with the resulting test suites, compared to performing mutation analysis with just the OTS. The "△" symbol in the table means that STICCER required a statistically significantly longer time to run than the OTS, while "▽" denotes the reverse. In general, STICCER is capable of substantial time savings for large schemas with large numbers of integrity constraints and test requirements. The largest saving is for the largest schema, *iTrust*, with a saving of approximately 16 minutes. Savings of 20 seconds are possible with *BookTown*, and *RiskIt*. For the smallest five schemas by the number of columns, STICCER performed statistically worse. In practice, however, that difference is negligible, always coming in at less than a second.

Table 5.5: Median Mutation Analysis Times (Seconds) for STICCER versus the other Reduction Techniques

The "▼" symbol means that significance tests show that the technique required a significantly longer time than STICCER, while "△" indicates the technique needed a significantly shorter time than STICCER.

	AVM-D				DOM-RND			
Schemas	STICCER	Random	Greedy	HGS	STICCER	Random	Greedy	HGS
ArtistSimilarity	0.05	*▼**0.07**	*▼**0.07**	*▼**0.06**	0.05	*▼**0.07**	*▼**0.07**	*▼**0.07**
ArtistTerm	0.22	*▼**0.38**	*▼**0.38**	*▼**0.33**	0.22	*▼**0.39**	*▼**0.37**	*▼**0.37**
BankAccount	0.19	*▼**0.32**	*▼**0.31**	*▼**0.28**	0.19	*▼**0.29**	*▼**0.28**	*▼**0.25**
BookTown	16.17	*▼**22.89**	*▼**22.05**	*▼**19.10**	10.85	*▼**22.37**	*▼**20.75**	*▼**19.11**
BrowserCookies	0.94	*▼**1.12**	*▼**1.02**	*▼**0.98**	0.73	*▼**1.27**	*▼**1.12**	*▼**1.04**
Cloc	0.07	*▼**0.17**	*▼**0.16**	*▼**0.15**	0.09	*▼**0.18**	*▼**0.17**	*▼**0.15**
CoffeeOrders	1.20	*▼**1.86**	*▼**1.88**	*▼**1.68**	1.05	*▼**1.81**	*▼**1.81**	*▼**1.57**
CustomerOrder	6.68	*△**6.47**	*▼**7.13**	*△**5.88**	4.56	*▼**6.32**	*▼**7.02**	*▼**5.68**
DellStore	1.93	*▼**5.27**	*▼**5.15**	*▼**4.68**	3.22	*▼**5.02**	*▼**4.71**	*▼**4.53**
Employee	0.13	*▼**0.21**	*▼**0.20**	*▼**0.21**	0.12	*▼**0.19**	*▼**0.16**	*▼**0.16**
Examination	1.39	*▼**2.34**	*▼**2.32**	*▼**2.16**	1.00	*▼**2.08**	*▼**1.80**	*▼**1.69**
Flights	0.61	*▼**0.85**	*▼**0.72**	*▼**0.71**	0.68	*▼**0.95**	*▼**0.87**	*▼**0.84**
FrenchTowns	1.46	*▼**1.59**	*▼**1.52**	*▼**1.55**	1.23	*▼**1.63**	*▼**1.62**	*▼**1.54**
Inventory	0.05	*▼**0.07**	*▼**0.07**	*▼**0.06**	0.04	*▼**0.07**	*▼**0.06**	*▼**0.06**
IsoFlav_R2	2.83	*▼**5.51**	*▼**5.24**	*▼**5.01**	2.72	*▼**5.04**	*▼**4.33**	*▼**4.04**
Iso3166	0.02	*▼**0.03**	*▼**0.03**	*▼**0.03**	0.03	*▼**0.03**	*▼**0.03**	*▼**0.03**
iTrust	959.31	*▼**1367.85**	*▼**1251.95**	*▼**1223.85**	428.57	*▼**1273.12**	*▼**1155.09**	*▼**1117.88**
JWhoisServer	5.62	*▼**5.88**	*▼**5.73**	*△**5.48**	3.88	*▼**5.81**	*▼**5.56**	*▼**5.29**
MozillaExtensions	6.62	*▼**12.35**	*▼**10.22**	*▼**12.74**	5.39	*▼**11.75**	*▼**9.89**	*▼**9.81**
MozillaPermissions	0.08	*▼**0.16**	*▼**0.15**	*▼**0.15**	0.07	*▼**0.14**	*▼**0.13**	*▼**0.12**
NistDML181	0.11	*▼**0.20**	*▼**0.19**	*▼**0.18**	0.12	*▼**0.22**	*▼**0.18**	*▼**0.18**
NistDML182	2.22	*▼**7.14**	*▼**6.50**	*▼**6.31**	2.43	*▼**7.54**	*▼**6.24**	*▼**5.85**
NistDML183	0.09	*▼**0.17**	*▼**0.17**	*▼**0.17**	0.10	*▼**0.18**	*▼**0.16**	*▼**0.17**
NistWeather	0.29	*▼**0.45**	*▼**0.41**	*▼**0.41**	0.25	*▼**0.46**	*▼**0.42**	*▼**0.41**
NistXTS748	0.04	*▼**0.06**	*▼**0.05**	*▼**0.05**	0.04	*▼**0.05**	*▼**0.05**	*▼**0.05**
NistXTS749	0.20	*▼**0.24**	*▼**0.24**	0.21	0.16	*▼**0.25**	*▼**0.22**	*▼**0.22**
Person	0.09	*▼**0.12**	▼**0.10**	*▼**0.10**	0.07	*▼**0.11**	*▼**0.11**	*▼**0.10**
Products	0.42	*▼**0.64**	*▼**0.61**	*▼**0.55**	0.44	*▼**0.68**	*▼**0.63**	*▼**0.59**
RiskIt	23.15	*▼**25.53**	*▼**25.79**	*△**22.77**	18.90	*▼**26.43**	*▼**24.60**	*▼**21.52**
StackOverflow	0.97	*▼**3.15**	*▼**3.00**	*▼**2.97**	1.42	*▼**2.73**	*▼**2.53**	*▼**2.48**
StudentResidence	0.20	*▼**0.28**	*▼**0.27**	*▼**0.26**	0.17	*▼**0.28**	*▼**0.25**	*▼**0.25**
UnixUsage	6.21	*▼**6.97**	*▼**7.52**	*▼**6.30**	3.83	*▼**7.57**	*▼**6.96**	*▼**5.91**
Usda	2.78	*▼**8.89**	*▼**8.60**	*▼**9.34**	4.15	*▼**9.72**	*▼**7.50**	*▼**7.33**
WordNet	1.86	*▼**2.45**	*▼**2.21**	*▼**2.07**	1.69	*▼**2.44**	*▼**2.28**	*▼**2.09**

Table 5.5 compares STICCER with the other reduction techniques. The results show that STICCER test suites generated with DOM-RND ran significantly faster than all reduced test suite by other techniques. STICCER was over 2.5 times faster than the other techniques for *iTrust* schema. STICCER was generally, but not always faster than other reduction techniques for tests suites generated by AVM-D. Since the reduction techniques were less successful at reducing AVM-D-generated test suites, there is less to differentiate their performance.

In conclusion for RQ3, the results show that, in general, test suites reduced by STICCER are fast to run compared to the OTS and those reduced by other techniques. For the largest schema, STICCER-reduced test suites were up to five times faster to run mutation analysis with, compared to the OTS.

5.7 Summary

Since many real-world database applications contain complex and rapidly evolving database schemas [35, 22, 133], there is the need for an efficient way to regression test these systems. While the automatically generated test suites created by tools like *SchemaAnalyst* [14] mitigate the challenges associated with manually testing a database schema's integrity constraints, the numerous, often interwoven, generated tests make repeated testing and test adequacy assessment time consuming. This chapter presents a test suite reduction technique, called **S**chema **T**est **I**ntegrity **C**onstraints **C**ombination for **E**fficient **R**eduction (STICCER), that systematically discards and merges redundant tests, creating a reduced test suite that is guaranteed to have the same coverage as the original one. STICCER advances the state-of-the-art in test suite reduction because, unlike traditional approaches, such as Greedy and HGS, it identifies and reduces both the overlap in test requirement coverage and the database state created by the tests.

Using 34 relational database schemas and test data created by two test generation methods, this chapter experimentally compared STICCER to greedy, HGS, and a Random method. These results show that STICCER significantly outperforms the other techniques at decreasing test suite size, while also lessening the overall number of database interactions (i.e., the SQL `INSERT` statements) performed by the tests. The results further reveal that the reduced test suites produced by STICCER are up to 5X faster than the original test suite and 2.5X faster than reduced suites created by greedy, HGS, and Random, often leading to significant decreases in mutation analysis time. STICCER's tests always preserve the coverage of the test suite and rarely lead to a drop in the mutation score, with a maximum decrease in fault detection of 3.2%.

Since HGS show to reduce test suites further than greedy, this can help the merging process to further reduce the test suite. Thus, STICCER integrated with HGS can produce a test suite that is more efficient than these evaluated in this chapter. Intuitively, reduced test suite are faster to inspect by testers making HGS reduced test suites more preferable for the evaluation process compared to other reduction methods. Because STICCER reduce and merge tests producing fewer equivalent but long tests that might require more effort to evaluate. Therefore, the next chapter will explore the HGS combined with STICCER, and conduct a human study to evaluate the testers' inspection efforts of differently reduced tests.

Chapter 6

Can Human Testers Effectively Inspect Reduced Versions of Automatically Generated SQL Test Suites?

The content of this chapter is based on the published work during this PhD in the 1st International Conference on Automation of Software Test (AST 2020) [163].

6.1 Introduction

The previous chapter introduced the STICCER reduction technique that was effective at reducing test suites using a merging mechanism. It worked by first reducing test suites using Greedy and subsequently merged with STICCER. This resulted with test suites that ran nearly 2.5 times faster than test suites reduced by traditional techniques, such HGS and Greedy. Also, the STICCER reduced test suites that ran up to 5 times faster than the original test suite. However, the results showed that the HGS technique reduced test suites further than the Greedy technique. Therefore, this chapter aims to present a new hybridisation of STICCER and HGS, instead of Greedy.

Many schema modifications force developers to test their changes and re-running the automatically generated tests, with follow-on steps that ensure the test suite's continued effectiveness by adequacy assessment through both coverage and mutation analysis [136]. Therefore, testers will reduce generated test suites to expedite regression testing and might inspect these tests that are either only reduced or merged, as stated in Section 2.2.4. Therefore, two questions raises: (1) Can STICCER further reduce HGS test suites? (2) Are shorter test cases reduced using HGS more effective for testers to inspect against the merged and longer test cases?

This chapter presents a variant STICCER technique, called STICCER-HGS, that merges the HGS reduced test suites. With this technique, the chapter will show two studies: (1) a computational study that evaluate both STICCER variants; (2) a human study that evaluate testers inspection effort on both HGS and STICCER-HGS reduced test suites. The latter study is motivated with STICCER-HGS's merged tests compared to only HGS reduced tests. One set of tests are long which we speculate that tester will have harder to inspect compared to only reduced short tests. Hence, conducting a human pilot study that evaluate the quantitative human oracle cost on HGS and STICCER-HGS reduced test suites.

Using 34 relational database schemas, two state-of-the-art test data generators, and the two hybridised and two traditional test suite reduction methods, this chapter's Computational Study finds that, while the hybridized methods outperform the stand-alone use of either Greedy or HGS, there is, surprisingly, no significant benefit to using HGS instead of Greedy in STICCER. Since this chapter's focus is on the benefits that may arise from combining HGS and STICCER, the Human Study asked 27 participants to act as testers who had to manually inspect test suites that had been reduced by either STICCER-HGS or HGS. This chapter's Human Study reveals that, compared to those produced by HGS, the reduced test suites made by STICCER-HGS help humans to complete test inspection tasks faster, but not more accurately. Along with confirming the benefits that accrue from hybridising STICCER with either greedy or HGS, this chapter's two studies suggest that, while test suite reduction may make certain testing tasks — like assessing test suite adequacy through mutation analysis — more efficient, it will not always benefit the humans testers who must inspect the reduced test suites.

In summary, this chapter's contributions are as follows:

1. A test suite reduction method, called STICCER-HGS, that takes advantage of shorter test suites and merge them.
2. A Computational experiment using 34 schemas and two state-of-the-art test data generators to compare the reduction effectiveness of STICCER-HGS against STICCER and other reduction technique.
3. A human study that compare testers inspect effort with STICCER-HGS and HGS reduced test suites.

To support the replication of this chapter's experimental results and to facilitate the testing of relational database schemas, the proposed techniques are implemented into *SchemaAnalyst* [145] and the procedures into scripts. Replication instructions are available in Appendix B.3.

6.2 The STICCER-HGS Approach

The HGS method in the previous chapter on average achieved a reduction of 46% and 50% for AVM-D and DOM-RND generated test suites, respectively. More effective than the greedy method that only achieved 43% and 48% reduction for both test data generators. Hence, the HGS may benefit the STICCER technique compared to the greedy method, merging shorter test suites. This introduces the STICCER-HGS variant that merges the reduced test suite with HGS. The technique is similar to the original variant, replacing the greedy method with the HGS technique, specifically in line 2 of Figure 5.3.

The following section will compare the reduction methods for their effectiveness, fault-finding capabilities, and duration of mutation analysis, answering the first three research questions. This is to ensure that the new variant is either equally or more effective than the original STICCER, preparing for the human study. Afterwards, two techniques that produced the shortest test suites will be evaluated, comparing the reduced and merged tests with a human study while answer the last two research questions.

6.3 The STICCER-HGS Computational Study

The aim of this computational study is to answer the following three research questions:

RQ1: Reduction Effectiveness. How does STICCER-HGS compare at reducing test suites to the original STICCER, HGS, and greedy?

RQ3: Reduction and Mutation Analysis Runtime. How does the fault-finding capability of test suites reduced by STICCER-HGS compare to those reduced by the original STICCER, HGS, and greedy?

RQ3: Reduction and Mutation Analysis Runtime. How does the overall time taken to (a) reduce test suites and then (b) perform mutation analysis on them compare when using either STICCER-HGS or the original STICCER as the test suite reduction technique?

6.3.1 Methodology

The *SchemaAnalyst* tool [14] was used to generate test suites with both DOM-RND and AVM-D for each of our subject schemas detailed in Appendix A. Following Chapter 5 experimental procedure, *SchemaAnalyst* was configured both test data generators to fulfill the “ClauseAICC+ANCC+AUCC” combination of coverage criteria (introduced in Section 2.3.5), with a termination criterion of 100,000 test data evaluations per test requirement (should test data not be found earlier than this limit). The *SchemaAnalyst* framework was also configured to generate test suites for the well-known SQLite DBMS, following the previous chapter procedure. The reason for using this combined coverage criterion was that it been reported as the strongest to detect seeded faults [7]. Since both DOM-RND and AVM-D are based on random number generation, *SchemaAnalyst* was set to repeat test generation 30 times. Then, STICCER-HGS, STICCER, HGS, and greedy are used to reduce each of the test suites, recording the execution time taken. Studying the adequacy assessment process for the reduced test suites, we next used *SchemaAnalyst* to run mutation analysis on each of them, applying Wright et al.’s mutation operators [101], again recording the time taken.

To answer RQ1, STICCER-HGS is compared at reducing the test suites generated by *SchemaAnalyst* with implementations of the original STICCER method (introduced in Section 5.4). Reduction effectiveness was calculated for test suite size and the number of `INSERT`s using Equation 5.1.

The results report the median values of this equation for both methods for the 30 test suites generated for each schema with the two test generation methods, and calculate statistical significance and effect sizes as detailed in Section 5.5.2.

To answer RQ2, the fault-finding capability of the reduced test suites using mutation analysis is investigated. For this, the mutation analysis techniques implemented into *SchemaAnalyst* is used, which adopt Wright et al.'s [101] mutation operators for integrity constraints. Calculating the mutation score for each reduced test suite as a percentage of mutants that are "killed" (i.e., detected) by the test suites. The scores will be then compared to the original test suites to evaluate the maintaining of fault-finding capabilities.

To answer RQ3, times needed to reduce each test suite is tracked, and the time needed to perform mutation analysis using the reduced suites.

This section's experiments were performed on the same Linux workstation as in Chapter 3 and Chapter 5. Similarly, we applied Chapter 5 statistical analysis in Section 5.5.2. Furthermore, this section's threats to validity and threats mitigations are the same as Section 5.5.3. To summarise, the first threat is the schema sets that may not generalize claims. To mitigate this threat, schemas in database testing research is used that are diverse, however it is difficult to obtain a set that is generalizable. Another threat is the stochastic behaviour and the timing measures of both the test data generators and mutation analysis which can be unpredictable. This is mitigated by repeating the experiments 30 times and the use of non-parametric hypothesis tests as recommended by Arcuri and Briand [142]. The implementation defects might be present in the code-base that is mitigated with applying unit tests.

It is also possible that the ordering of test cases passed to STICCER for merging could be a considered a potential threat to validity, as this could affect which test cases are merged with one another, and hence the results we obtain. Yet, we experimented with randomizing and reversing the order of test cases passed to STICCER from HGS ("irreplaceable" tests first) and Greedy (most test-requirement-covering tests first), but did not observe significant differences in the results. Therefore, we continued to use the default order of tests provided by the reduction techniques prior to merging.

Table 6.1: Median Reduction Effectiveness for Test Suites

Reduction effectiveness using Equation 5.1 and higher–is-better.

	AVM-D								DOM-RND							
Schema	STICCER-HGS		HGS		STICCER		greedy		STICCER-HGS		HGS		STICCER		greedy	
ArtistSimilarity	63%	(7/19)	*▼**42%**	(11/19)	63%	(7/19)	*▼**32%**	(13/19)	63%	(7/19)	*▼**32%**	(13/19)	63%	(7/19)	*▼**37%**	(12/19)
ArtistTerm	65%	(15/43)	*▼**37%**	(27/43)	65%	(15/43)	*▼**30%**	(30/43)	63%	(16/43)	*▼**30%**	(30/43)	*Δ**65%**	(15/43)	*▼**30%**	(30/43)
BankAccount	68%	(12/37)	*▼**43%**	(21/37)	68%	(12/37)	*▼**38%**	(23/37)	70%	(11/37)	*▼**49%**	(19/37)	*▼**70%**	(11/37)	*▼**43%**	(21/37)
BookTown	63%	(100/269)	*▼**46%**	(144/269)	*▼**58%**	(113/269)	*▼**38%**	(167/269)	65%	(94/269)	*▼**49%**	(138/269)	*Δ**68%**	(87/269)	*▼**42%**	(156/269)
BrowserCookies	63%	(26/71)	*▼**59%**	(29/71)	*▼**62%**	(27/71)	*▼**56%**	(31/71)	77%	(16/71)	*▼**56%**	(31/71)	76%	(17/71)	*▼**55%**	(32/71)
Cloc	90%	(4/40)	*▼**50%**	(20/40)	90%	(4/40)	*▼**48%**	(21/40)	75%	(10/40)	*▼**57%**	(17/40)	*Δ**85%**	(6/40)	*▼**51%**	(20/40)
CoffeeOrders	64%	(32/90)	*▼**42%**	(52/90)	64%	(32/90)	*▼**41%**	(53/90)	70%	(27/90)	*▼**47%**	(48/90)	*Δ**74%**	(23/90)	*▼**44%**	(50/90)
CustomerOrder	44%	(71/126)	*▼**37%**	(79/126)	*▼**40%**	(76/126)	*▼**33%**	(84/126)	64%	(45/126)	*▼**39%**	(77/126)	*▼**63%**	(46/126)	*▼**35%**	(82/126)
DellStore	86%	(24/177)	*▼**38%**	(110/177)	86%	(24/177)	*▼**35%**	(115/177)	67%	(59/177)	*▼**42%**	(102/177)	*Δ**71%**	(52/177)	*▼**41%**	(105/177)
Employee	71%	(11/38)	*▼**50%**	(19/38)	*Δ**74%**	(10/38)	*▼**50%**	(19/38)	82%	(7/38)	*▼**61%**	(15/38)	80%	(8/38)	*▼**61%**	(15/38)
Examination	72%	(30/107)	*▼**52%**	(51/107)	72%	(30/107)	*▼**51%**	(52/107)	85%	(16/107)	*▼**64%**	(38/107)	*Δ**87%**	(14/107)	*▼**64%**	(38/107)
Flights	71%	(18/62)	*▼**58%**	(26/62)	*▼**69%**	(19/62)	*▼**58%**	(26/62)	71%	(18/62)	*▼**53%**	(29/62)	71%	(18/62)	*▼**52%**	(30/62)
FrenchTowns	38%	(33/53)	*▼**34%**	(35/53)	*Δ**40%**	(32/53)	*▼**36%**	(34/53)	58%	(22/53)	*▼**34%**	(35/53)	60%	(21/53)	*▼**32%**	(36/53)
Inventory	72%	(5/18)	*▼**44%**	(10/18)	*▼**67%**	(6/18)	*▼**39%**	(11/18)	72%	(5/18)	*▼**56%**	(8/18)	78%	(4/18)	*▼**56%**	(8/18)
IsoFlav_R2	76%	(43/177)	*▼**50%**	(88/177)	*▼**75%**	(45/177)	*▼**49%**	(90/177)	78%	(39/177)	*▼**62%**	(66/177)	*Δ**81%**	(34/177)	*▼**60%**	(70/177)
Iso3166	58%	(5/12)	*▼**33%**	(8/12)	58%	(5/12)	*▼**33%**	(8/12)	58%	(5/12)	*▼**42%**	(7/12)	58%	(5/12)	*▼**33%**	(8/12)
iTrust	58%	(631/1517)	*▼**44%**	(847/1517)	*▼**57%**	(646/1517)	*▼**43%**	(872/1517)	80%	(297/1517)	*▼**50%**	(754/1517)	*Δ**85%**	(235/1517)	*▼**49%**	(776/1517)
JWhoisServer	39%	(97/158)	*▼**35%**	(103/158)	*▼**37%**	(100/158)	*▼**33%**	(106/158)	65%	(56/158)	*▼**37%**	(99/158)	*Δ**70%**	(48/158)	*▼**35%**	(103/158)
MozillaExtensions	75%	(57/229)	*▼**50%**	(115/229)	▼**75%**	(57/229)	*▼**60%**	(92/229)	82%	(42/229)	*▼**64%**	(83/229)	*Δ**85%**	(35/229)	*▼**63%**	(84/229)
MozillaPermissions	73%	(9/33)	*▼**48%**	(17/33)	73%	(9/33)	*▼**48%**	(17/33)	79%	(7/33)	*▼**64%**	(12/33)	*Δ**88%**	(4/33)	*▼**58%**	(14/33)
NistDML181	82%	(7/38)	*▼**53%**	(18/38)	*▼**79%**	(8/38)	*▼**53%**	(18/38)	79%	(8/38)	*▼**58%**	(16/38)	▼**76%**	(9/38)	*▼**55%**	(17/38)
NistDML182	90%	(19/190)	*▼**57%**	(81/190)	*▼**89%**	(20/190)	*▼**57%**	(82/190)	88%	(22/190)	*▼**62%**	(73/190)	Δ**89%**	(21/190)	*▼**60%**	(76/190)
NistDML183	82%	(6/34)	*▼**53%**	(16/34)	82%	(6/34)	*▼**47%**	(18/34)	76%	(8/34)	*▼**53%**	(16/34)	76%	(8/34)	*▼**50%**	(17/34)
NistWeather	64%	(20/56)	*▼**43%**	(32/56)	▼**64%**	(20/56)	*▼**45%**	(31/56)	80%	(11/56)	*▼**46%**	(30/56)	*Δ**82%**	(10/56)	*▼**45%**	(31/56)
NistXTS748	62%	(6/16)	*▼**44%**	(9/16)	62%	(6/16)	*▼**38%**	(10/16)	75%	(4/16)	*▼**50%**	(8/16)	69%	(5/16)	*▼**50%**	(8/16)
NistXTS749	63%	(13/35)	*▼**51%**	(17/35)	*▼**57%**	(15/35)	*▼**46%**	(19/35)	69%	(11/35)	*▼**49%**	(18/35)	69%	(11/35)	*▼**49%**	(18/35)
Person	50%	(10/20)	*▼**40%**	(12/20)	50%	(10/20)	*▼**40%**	(12/20)	70%	(6/20)	*▼**35%**	(13/20)	*Δ**80%**	(4/20)	*▼**30%**	(14/20)
Products	67%	(17/52)	*▼**48%**	(27/52)	67%	(17/52)	*▼**44%**	(29/52)	70%	(16/52)	*▼**46%**	(28/52)	69%	(16/52)	*▼**44%**	(29/52)
RiskIt	56%	(110/250)	*▼**48%**	(130/250)	*▼**51%**	(122/250)	*▼**43%**	(142/250)	64%	(91/250)	*▼**52%**	(120/250)	63%	(92/250)	*▼**48%**	(129/250)
StackOverflow	93%	(12/171)	*▼**50%**	(86/171)	93%	(12/171)	*▼**48%**	(89/171)	78%	(38/171)	*▼**60%**	(68/171)	*Δ**84%**	(28/171)	*▼**60%**	(68/171)
StudentResidence	62%	(13/34)	*▼**44%**	(19/34)	62%	(13/34)	*▼**41%**	(20/34)	74%	(9/34)	*▼**47%**	(18/34)	74%	(9/34)	*▼**47%**	(18/34)
UnixUsage	56%	(64/147)	*▼**47%**	(78/147)	*▼**52%**	(70/147)	*▼**43%**	(84/147)	71%	(42/147)	*▼**50%**	(73/147)	*Δ**73%**	(40/147)	*▼**48%**	(76/147)
Usda	88%	(30/247)	*▼**44%**	(139/247)	88%	(30/247)	*▼**40%**	(147/247)	72%	(70/247)	*▼**51%**	(121/247)	*Δ**78%**	(54/247)	*▼**49%**	(125/247)
WordNet	59%	(48/118)	*▼**44%**	(66/118)	*▼**58%**	(50/118)	*▼**42%**	(69/118)	60%	(47/118)	*▼**44%**	(66/118)	*Δ**64%**	(43/118)	*▼**40%**	(71/118)
Minimum	38%		33%		37%		30%		58%		30%		58%		30%	
Average	67%		46%		66%		43%		72%		50%		74%		48%	
Maximum	93%		59%		93%		60%		88%		64%		89%		64%	

Table 6.2: Median Reduction Effectiveness for SQL INSERT statements (Database Interactions) in the Reduced Test Suites

Reduction effectiveness of `INSERT` statements were higher–is-better.

	AVM-D								DOM-RND							
Schema	STICCER-HGS		HGS		STICCER		greedy		STICCER-HGS		HGS		STICCER		greedy	
ArtistSimilarity	52%	(21/44)	*▼**45%**	(24/44)	▼**48%**	(23/44)	*▼**34%**	(29/44)	50%	(22/44)	*▼**35%**	(28/44)	50%	(22/44)	*▼**39%**	(27/44)
ArtistTerm	59%	(51/124)	*▼**45%**	(68/124)	▼**56%**	(54/124)	*▼**36%**	(79/124)	54%	(56/124)	*▼**36%**	(79/124)	*△**56%**	(54/124)	*▼**36%**	(79/124)
BankAccount	59%	(33/80)	*▼**44%**	(45/80)	▼**56%**	(35/80)	*▼**38%**	(50/80)	57%	(34/80)	*▼**50%**	(40/80)	57%	(34/80)	*▼**44%**	(44/80)
BookTown	52%	(194/403)	*▼**47%**	(215/403)	*▼**44%**	(225/403)	*▼**38%**	(250/403)	51%	(197/403)	*▼**48%**	(208/403)	*▼**49%**	(206/403)	*▼**42%**	(233/403)
BrowserCookies	65%	(61/175)	*▼**64%**	(63/175)	64%	(63/175)	*▼**62%**	(66/175)	69%	(54/175)	*▼**60%**	(70/175)	69%	(55/175)	*▼**59%**	(71/175)
Cloc	63%	(22/60)	*▼**50%**	(30/60)	▼**62%**	(23/60)	*▼**48%**	(31/60)	58%	(25/60)	*▼**55%**	(27/60)	*△**61%**	(24/60)	*▼**48%**	(31/60)
CoffeeOrders	61%	(106/273)	*▼**48%**	(143/273)	▼**61%**	(107/273)	*▼**45%**	(149/273)	64%	(98/273)	*▼**51%**	(132/273)	65%	(96/273)	*▼**49%**	(139/273)
CustomerOrder	43%	(273/475)	*▼**39%**	(288/475)	*▼**39%**	(290/475)	*▼**36%**	(305/475)	59%	(196/475)	*▼**41%**	(279/475)	*▼**56%**	(208/475)	*▼**36%**	(302/475)
DellStore	58%	(118/281)	*▼**42%**	(162/281)	▼**56%**	(123/281)	*▼**39%**	(172/281)	48%	(145/281)	*▼**47%**	(150/281)	*△**50%**	(141/281)	*▼**44%**	(156/281)
Employee	58%	(22/53)	*▼**47%**	(28/53)	*△**60%**	(21/53)	*▼**47%**	(28/53)	62%	(20/53)	*▼**57%**	(23/53)	62%	(20/53)	*▼**57%**	(23/53)
Examination	66%	(77/229)	*▼**52%**	(111/229)	▼**66%**	(78/229)	*▼**50%**	(114/229)	73%	(62/229)	*▼**64%**	(83/229)	*△**74%**	(60/229)	*▼**63%**	(86/229)
Flights	71%	(40/137)	*▼**66%**	(46/137)	*▼**70%**	(41/137)	*▼**66%**	(46/137)	58%	(57/137)	*▼**55%**	(62/137)	58%	(58/137)	*▼**55%**	(62/137)
FrenchTowns	47%	(85/161)	*▼**43%**	(91/161)	*△**50%**	(81/161)	*▼**47%**	(86/161)	53%	(76/161)	*▼**43%**	(92/161)	52%	(77/161)	*▼**40%**	(96/161)
Inventory	57%	(12/28)	*▼**43%**	(16/28)	*▼**54%**	(13/28)	*▼**39%**	(17/28)	61%	(11/28)	*▼**54%**	(13/28)	▼**57%**	(12/28)	*▼**54%**	(13/28)
IsoFlav_R2	63%	(102/274)	*▼**50%**	(136/274)	*▼**62%**	(104/274)	*▼**50%**	(136/274)	64%	(100/274)	*▼**59%**	(111/274)	64%	(100/274)	*▼**58%**	(116/274)
Iso3166	47%	(10/19)	*▼**37%**	(12/19)	47%	(10/19)	*▼**37%**	(12/19)	47%	(10/19)	47%	(10/19)	▼**42%**	(11/19)	*▼**37%**	(12/19)
iTrust	57%	(946/2204)	*▼**50%**	(1103/2204)	*▼**56%**	(978/2204)	*▼**48%**	(1142/2204)	56%	(978/2204)	*▼**52%**	(1064/2204)	*△**57%**	(940/2204)	*▼**50%**	(1101/2204)
JWhoisServer	40%	(153/256)	40%	(153/256)	*▼**38%**	(158/256)	*▼**38%**	(158/256)	45%	(141/256)	*▼**43%**	(145/256)	*△**47%**	(136/256)	*▼**41%**	(152/256)
MozillaExtensions	63%	(130/356)	*▼**50%**	(179/356)	△**71%**	(105/356)	63%	(130/356)	67%	(119/356)	*▼**61%**	(140/356)	*△**69%**	(110/356)	*▼**60%**	(144/356)
MozillaPermissions	62%	(19/50)	*▼**48%**	(26/50)	62%	(19/50)	*▼**48%**	(26/50)	62%	(19/50)	*▼**60%**	(20/50)	*△**66%**	(17/50)	*▼**54%**	(23/50)
NistDML181	68%	(25/78)	*▼**53%**	(37/78)	*▼**68%**	(25/78)	*▼**55%**	(35/78)	68%	(25/78)	*▼**56%**	(34/78)	▼**65%**	(28/78)	*▼**54%**	(36/78)
NistDML182	74%	(100/384)	*▼**61%**	(150/384)	*▼**73%**	(102/384)	*▼**61%**	(151/384)	75%	(97/384)	*▼**63%**	(144/384)	74%	(98/384)	*▼**61%**	(149/384)
NistDML183	68%	(22/68)	*▼**53%**	(32/68)	▼**65%**	(24/68)	*▼**46%**	(37/68)	65%	(24/68)	*▼**51%**	(33/68)	62%	(26/68)	*▼**50%**	(34/68)
NistWeather	60%	(48/120)	*▼**49%**	(61/120)	△**61%**	(47/120)	*▼**52%**	(58/120)	63%	(44/120)	*▼**52%**	(58/120)	63%	(44/120)	*▼**51%**	(58/120)
NistXTS748	48%	(12/23)	*▼**39%**	(14/23)	48%	(12/23)	*▼**35%**	(15/23)	57%	(10/23)	*▼**48%**	(12/23)	52%	(11/23)	*▼**48%**	(12/23)
NistXTS749	60%	(29/73)	*▼**53%**	(34/73)	*▼**53%**	(34/73)	*▼**47%**	(39/73)	59%	(30/73)	*▼**49%**	(37/73)	58%	(30/73)	*▼**49%**	(37/73)
Person	53%	(14/30)	*▼**50%**	(15/30)	53%	(14/30)	*▼**50%**	(15/30)	43%	(17/30)	43%	(17/30)	*△**47%**	(16/30)	*▼**37%**	(19/30)
Products	67%	(48/144)	*▼**56%**	(63/144)	▼**65%**	(50/144)	*▼**53%**	(68/144)	63%	(53/144)	*▼**53%**	(68/144)	▼**62%**	(54/144)	*▼**51%**	(70/144)
RiskIt	55%	(312/687)	*▼**51%**	(336/687)	*▼**50%**	(342/687)	*▼**47%**	(366/687)	57%	(297/687)	*▼**53%**	(322/687)	▼**54%**	(318/687)	*▼**50%**	(346/687)
StackOverflow	65%	(90/257)	*▼**50%**	(129/257)	▼**64%**	(93/257)	*▼**48%**	(134/257)	62%	(98/257)	*▼**57%**	(110/257)	*△**66%**	(88/257)	*▼**57%**	(111/257)
StudentResidence	57%	(31/72)	*▼**46%**	(39/72)	▼**56%**	(32/72)	*▼**42%**	(42/72)	60%	(29/72)	*▼**49%**	(37/72)	60%	(29/72)	*▼**49%**	(37/72)
UnixUsage	61%	(232/595)	*▼**51%**	(293/595)	*▼**58%**	(252/595)	*▼**47%**	(313/595)	69%	(187/595)	*▼**54%**	(274/595)	*△**70%**	(178/595)	*▼**52%**	(287/595)
Usda	61%	(149/381)	*▼**46%**	(206/381)	▼**59%**	(157/381)	*▼**42%**	(220/381)	55%	(171/381)	*▼**52%**	(181/381)	*△**59%**	(157/381)	*▼**50%**	(190/381)
WordNet	52%	(93/192)	*▼**49%**	(98/192)	*▼**50%**	(96/192)	*▼**47%**	(102/192)	50%	(96/192)	*▼**49%**	(98/192)	49%	(97/192)	*▼**45%**	(106/192)
Minimum	40%		37%		38%		34%		43%		35%		42%		36%	
Average	59%		49%		57%		46%		59%		51%		59%		49%	
Maximum	74%		66%		73%		66%		75%		64%		74%		63%	

6.3.2 Answering RQ1 through RQ3

RQ1: Reduction Effectiveness

Tables 6.1 and 6.2 show the median reduction effectiveness of each technique at decreasing the number of test cases for each schema and the total number of statements (i.e., database `INSERT`s) in the test cases of the test suites, respectively. In both table, the "∇" denote that the reduction techniques score was significantly less reduced than the STICCER-HGS, while "Δ" indicates STICCER-HGS is significantly higher. The "$*$" symbol denotes a large effect size between STICCER-HGS and other techniques. The numbers in brackets indicate the median tests/`INSERT`s reduced within over the unreduced tests/`INSERT`s.

Both tables report effectiveness for test suites generated by AVM-D and DOM-RND, because, as the tables reveal, the reduction techniques vary in performance depending on which test generation technique was initially used. Overall, four different trends are observed in the two tables, which will be explained next.

Firstly, STICCER-HGS significantly outperforms HGS and greedy, regardless of initial test generation technique, just as the original STICCER did in the previous chapter's study. Table 6.1 shows that STICCER-HGS is significantly better than HGS and greedy at reducing the number of test cases for all schemas, while Table 6.2 shows that STICCER-HGS also significantly reduces the total number of statements in the tests suites compared to HGS and greedy, for all but a few schemas.

Secondly, STICCER-HGS is, overall, more effective at reducing DOM-RND-generated test suites than those made by AVM-D. Table 6.1 shows an overall reduction mean of 72% with DOM-RND-generated test suites, compared to 67% with AVM-D-generated suites. As we previously observed in Chapter 5, the same is true for the original STICCER, where the averages are 74% with DOM-RND compared to 66% with AVM-D. This phenomenon is explained and centred on the data values that each test data generator typically generates. AVM-D repeats "default" values such as empty strings and zero numerical values, aiming to keep test cases as simple as possible. However, this frustrates technique's attempts to merge `INSERT` statements,

Successful Merge of DOMINO Test Cases

	id	last_name	first_name	gender	date_of_birth	
t_1	**-458**	'ada'	'djd'	'Male'	'2008-06-10'	✓
t_2	**0**	'ib'	'edvbewwyg'	'Other'	'1992-03-17'	✓

Unsuccessful Merge of AVM-D Test Cases

	id	last_name	first_name	gender	date_of_birth	
t_1	**0**	''	''	'Male'	'2000-01-01'	✓
t_2	**0**	''	''	'Other'	'2000-01-01'	✗

Figure 6.1: STICCER's Attempts to Merge Test Cases

since the use of the same values across different test cases can inadvertently trigger primary key and `UNIQUE` constraint violations when two tests are combined. Figure 6.1 illustrates this phenomenon with an example. One of the test requirements that needs to be preserved by the merged test case in this instance are unique values for the `gender` field. Yet, the re-use of zero as an `id` value for the two tests that are attempting to merge in the AVM-D case results in a primary key violation. As such, the merged test case is not equivalent to the two original test cases, where the database state would have been reset between their execution.

The issue of test case diversity also helps to explain the third and fourth trends that we observe: STICCER-HGS is better, overall, at reducing AVM-D-generated test suites compared to the original STICCER— but conversely, the original STICCER is better, overall, at reducing DOMINO-generated test suites. Both of these phenomena are seen in the summary averages of Tables 6.1 and 6.2 — and also when comparing the respective number of schemas STICCER-HGS is significantly better at reducing compared to the original STICCER, and vice versa. In the AVM-D case, its choice of repetitious values hinder merging, resulting in the ultimate winner being strongly correlated to the effectiveness of the original reduction technique used — that is, HGS in the case of STICCER-HGS, which is more effective than greedy, used by the original STICCER. However, the merging technique can work more effectively with the diverse test cases generated by DOM-RND, and furthermore, it seems that the larger reduced test suites supplied by Greedy add to this diversity, allowing the merge mechanism to operate more effectively.

Hence, the original STICCER performs significantly better than STICCER-HGS in more cases than it does not for DOM-RND-generated test suites. In the cases that it does not, STICCER-HGS has the advantage of leveraging the more effective reduction provided by HGS. The "lift" of diversity that the original STICCER gets from less effective Greedy reduction can be seen for the AVM-D-generated test suites also, resulting in STICCER-HGS not being significantly better for every database schema.

The *BookTown* database schema provides a good illustration of both of these two trends. As shown in Table 6.1, the unreduced test suite has 269 test cases, which, in the AVM-D case are reduced to 144 and 167 test cases by HGS and Greedy respectively, and then further to 100 and 113 test cases following merging. The STICCER technique can reduce the test suite by more test cases in its merging phase for the original greedy-STICCER (54, as opposed to 44 achieved by STICCER-HGS), but the initial advantage given to STICCER-HGS by virtue of using HGS for reduction prior to merging is not completely overturned. Conversely, in the DOM-RND case, the original test suite size is reduced to 138 and 156 test cases by HGS and Greedy, respectively. However, because of the larger, more diverse pool of test cases produced by DOM-RND, the original STICCER technique overturns the initial advantage of HGS, reducing the test suite down to a final size of 87 test cases, as opposed to 94 for STICCER-HGS.

In conclusion for RQ1, like the original STICCER before it, STICCER-HGS significantly outperforms both HGS and greedy. The results show that STICCER-HGS is more effective with test suites generated using AVM-D, while the original STICCER is more effective for test suites generated with DOMINO. In general, STICCER's merging is more effective with the diverse test data values in DOM-RND-generated test cases, and works better with the slightly larger pool of test cases that Greedy tends to provide to the test merging mechanism.

Table 6.3: Median Mutation Scores

Percentages of detected mutants by the test suites. All the reduced test suites are compared to original, unreduced test suites (OTS).

	AVM-D					DOM-RND				
Schemas	OTS	STICCER-HGS	STICCER	Greedy	HGS	OTS	STICCER-HGS	STICCER	Greedy	HGS
BrowserCookies	86.5	*▼**86.0**	▼**86.5**	▼**86.5**	*▼**86.2**	96.6	96.6	96.6	96.6	96.6
FrenchTowns	83.3	*▼**80.3**	*▼**80.3**	*▼**80.3**	*▼**81.1**	95.5	95.5	95.5	95.5	95.5
iTrust	83.6	*▼**83.6**	*▼**83.6**	*▼**83.6**	*▼**83.6**	99.2	99.1	99.2	99.2	99.1
NistWeather	93.8	93.8	*▼**90.6**	*▼**90.6**	93.8	100.0	100.0	100.0	100.0	100.0
NistXTS749	92.0	*▼**88.0**	92.0	92.0	*▼**88.0**	94.0	94.0	94.0	94.0	94.0
RiskIt	89.3	*▼**88.8**	89.3	89.3	*▼**88.8**	99.5	99.5	99.5	99.5	99.5
UnixUsage	98.2	*▼**97.3**	98.2	98.2	*▼**97.3**	100.0	100.0	100.0	100.0	100.0
WordNet	87.4	*▼**86.3**	*▼**86.3**	*▼**86.3**	*▼**86.3**	99.0	99.0	99.0	99.0	99.0

RQ2: Fault Finding Capability

Table 6.3 show the median mutation scores for all techniques compared to the original test suite. The "∇" symbol in the table indicate that the reduced tests are significantly obtained lower mutation score compared to the OTS.

Similar to the original STICCER results in Chapter 5, the differences were only observed for eight of the two schema and experienced with test suites generated by AVM-D. The rest of schemas are redacted as they show no statistical significant differences.

In all cases, DOM-RND test suites are more robust for reduction, likely because of the diversity of test values that it generates. The results also show that both STICCER variants are identical with DOM-RND.

AVM-D-generated test suites, without the benefit of the same extent of diversity, did suffer in decreases in mutation score after reduction. AVM-D-generated and STICCER-HGS-reduced test suites received significantly worse mutation scores for seven schemas (each accompanied by a large effect size) than the original test suite, although the differences were not greater than 4%. Both STICCER's variants obtained similar mutation scores to their baseline reduction techniques (i.e., Greedy and HGS) against OTS. That is, the original STICCER scored significantly worse than the OTS for five schemas the same as greedy with equal scores. STICCER-HGS and HGS were significantly worse than OTS with equal in the number of schema with equal scores in seven of the eight. The *FrenchTowns* test suite reduced using STICCER-

HGS scored 0.8% worse than the HGS, with a p-*value* of 2.08×10^{-10} and large effect size. This is due to removing `INSERT`s while merging and lower test data diversity. Obtaining significantly lower fault detection of exchanging `FOREIGN KEY` and `UNIQUE` constraints.

In conclusion for RQ2, DOM-RND-generated test suites did not change mutation score following reduction. AVM-D-generated suites did incur decreased scores, but only for seven schemas and not $> 4\%$.

Table 6.4: Median Reduction and Mutation Times (Seconds) for STICCER-HGS versus the original STICCER

Times for STICCER-HGS and STICCER are broken down into “RT” (reduction time), and “MT” (mutation analysis time), with the MTs and totals statistically compared.

	AVM-D						DOM-RND					
	STICCER			STICCER-HGS			STICCER			STICCER-HGS		
Schemas	RT	MT	Total	RT	MT	Total	RT	MT	Total	RT	MT	Total
ArtistSimilarity	0.12	0.05	0.17	0.04	0.05	*▼**0.09**	0.13	0.05	0.18	0.05	0.05	*▼**0.10**
ArtistTerm	0.14	0.22	0.37	0.06	0.22	*▼**0.29**	0.14	0.22	0.36	0.05	0.23	*▼**0.28**
BankAccount	0.17	0.19	0.36	0.09	▼**0.18**	*▼**0.27**	0.16	0.19	0.34	0.08	△**0.20**	*▼**0.28**
BookTown	0.94	16.17	17.11	0.90	*▼**14.20**	*▼**15.10**	1.07	10.85	11.92	1.04	*△**11.46**	*△**12.50**
BrowserCookies	0.37	0.94	1.31	0.34	0.92	1.26	0.37	0.73	1.09	0.38	0.72	1.09
Cloc	0.15	0.07	0.22	0.07	0.07	*▼**0.14**	0.15	0.09	0.25	0.08	*△**0.12**	*▼**0.20**
CoffeeOrders	0.31	1.20	1.51	0.25	1.21	*▼**1.46**	0.29	1.05	1.34	0.22	*△**1.14**	1.35
CustomerOrder	0.80	6.68	7.47	0.70	*▼**6.22**	*▼**6.92**	0.84	4.56	5.40	0.76	*▼**4.35**	*▼**5.10**
DellStore	1.42	1.93	3.35	1.68	▼**1.86**	*△**3.54**	1.46	3.22	4.68	1.75	*△**3.37**	*△**5.12**
Employee	0.20	0.13	0.32	0.15	*△**0.14**	*▼**0.29**	0.18	0.12	0.30	0.14	0.11	*▼**0.25**
Examination	1.06	1.39	2.45	1.72	1.37	*△**3.09**	1.11	1.00	2.12	1.73	△**1.06**	*△**2.79**
Flights	0.30	0.61	0.91	0.20	0.61	▼**0.81**	0.41	0.68	1.09	0.37	△**0.71**	1.08
FrenchTowns	0.23	1.46	1.69	0.12	△**1.51**	1.62	0.23	1.23	1.46	0.14	1.23	*▼**1.37**
Inventory	0.12	0.05	0.17	0.04	*▼**0.04**	*▼**0.08**	0.12	0.04	0.17	0.05	0.04	*▼**0.09**
IsoFlav_R2	0.70	2.83	3.52	0.88	2.82	*△**3.70**	0.75	2.72	3.47	0.96	2.79	*△**3.75**
Iso3166	0.11	0.02	0.13	0.03	0.02	*▼**0.06**	0.11	0.03	0.13	0.04	0.03	*▼**0.06**
iTrust	150.17	959.31	1109.48	653.16	*▼**936.61**	*△**1589.77**	157.23	428.57	585.80	634.77	*△**522.86**	*△**1157.63**
JWhoisServer	1.64	5.62	7.25	1.90	*▼**5.39**	7.29	1.86	3.88	5.74	2.11	*△**4.28**	*△**6.39**
MozillaExtensions	2.02	6.62	8.64	5.16	*△**6.95**	*△**12.12**	2.38	5.39	7.78	5.26	*△**6.36**	*△**11.62**
MozillaPermissions	0.15	0.08	0.23	0.09	*△**0.10**	*▼**0.18**	0.15	0.07	0.22	0.08	*△**0.10**	*▼**0.18**
NistDML181	0.15	0.11	0.26	0.07	*▼**0.10**	*▼**0.18**	0.15	0.12	0.28	0.08	*▼**0.12**	*▼**0.20**
NistDML182	1.93	2.22	4.15	3.64	*△**2.43**	*△**6.07**	2.00	2.43	4.43	3.66	*△**2.59**	*△**6.25**
NistDML183	0.14	0.09	0.23	0.06	▼**0.09**	*▼**0.15**	0.14	0.10	0.25	0.07	0.10	*▼**0.17**
NistWeather	0.22	0.29	0.51	0.15	*△**0.30**	▼**0.45**	0.23	0.25	0.48	0.18	*△**0.26**	▼**0.44**
NistXTS748	0.12	0.04	0.16	0.04	*△**0.04**	*▼**0.08**	0.11	0.04	0.15	0.04	0.04	*▼**0.08**
NistXTS749	0.17	0.20	0.37	0.10	*▼**0.18**	*▼**0.28**	0.17	0.16	0.33	0.08	0.17	*▼**0.25**
Person	0.10	0.09	0.19	0.03	0.09	*▼**0.12**	0.15	0.07	0.22	0.08	*△**0.08**	*▼**0.16**
Products	0.14	0.42	0.57	0.08	0.42	▼**0.50**	0.18	0.44	0.62	0.12	0.44	*▼**0.56**
RiskIt	1.41	23.15	24.57	1.85	*▼**21.23**	*▼**23.08**	1.59	18.90	20.49	1.93	*▼**18.22**	▼**20.15**
StackOverflow	0.93	0.97	1.90	1.34	*▼**0.93**	*△**2.27**	1.16	1.42	2.58	1.51	*△**1.74**	*△**3.25**
StudentResidence	0.16	0.20	0.36	0.09	0.20	*▼**0.29**	0.14	0.17	0.32	0.07	0.17	*▼**0.24**
UnixUsage	0.92	6.21	7.13	1.04	*▼**5.69**	*▼**6.73**	0.95	3.83	4.79	1.07	*△**4.15**	*△**5.22**
Usda	1.39	2.78	4.17	1.88	*▼**2.71**	*△**4.59**	1.52	4.15	5.68	2.02	*△**5.31**	*△**7.33**
WordNet	0.37	1.86	2.23	0.34	*▼**1.77**	*▼**2.11**	0.38	1.69	2.07	0.29	*△**1.77**	2.07

RQ3: Reduction and Mutation Analysis Runtime

Table 6.4 shows the median mutation and reduction time for both STICCER-HGS and the original STICCER. The “△” symbol in the table means that STICCER-HGS required a statistically significantly longer time to run than the STICCER, while “∇” denotes the reverse.

The results show many significant differences in times recorded for STICCER-HGS and the original STICCER, the vast majority only correspond to a couple of seconds, and therefore are almost practically negligible.

The exception to this is the *iTrust* schema, which has the largest original test suite of 1517 test cases. Here, the overheads of the additional algorithmic complexity of HGS compared to Greedy are evident. HGS took a median of 11 minutes to reduce the AVM-D-generated test suites for the *iTrust* schema, compared to only 2 minutes with greedy reduction. As shown by Table **??**, following merging this results in smaller AVM-D-generated test suites on average for STICCER-HGS compared to the original STICCER (631 test cases as opposed to 646), but the DOM-RND-generated test suites are larger (297 as opposed to 231). Unsurprisingly, mutation analysis times follow the reduced test suite sizes, since the larger the test suite, the more work mutation analysis has to do. Overall, the additional time taken by HGS for the AVM-D-generated test suites is not sufficiently recovered in mutation analysis for the smaller suites of STICCER-HGS, resulting in the original STICCER recording a significantly faster time with AVM-D and DOM-RND test suites.

In conclusion for RQ3, although our experiments record many significant differences in timing, they are almost negligible in practical terms, except for the largest schema, *iTrust*. For this schema, STICCER-HGS was significantly slower for both AVM-D and the DOM-RND-generated test suites. In the AVM-D case, STICCER-HGS produces smaller test suites, but the additional time HGS needs to do this is not recovered in the savings made by mutation analysis.

Overall Conclusions of the Computational Study. The evidence suggests that STICCER's merging mechanism works better with the diverse DOM-RND-generated tests, and the slightly larger set of tests to choose from that arise from using Greedy. Yet, the results for each schema are more nuanced. For some schemas, the more heavily reduced test suites produced by HGS more than outweigh a slightly less efficient secondary merging phase for STICCER-HGS, particularly with those test suites generated by AVM-D.

The results of mutation analysis show a slight degradation of mutation scores for test suites initially generated by AVM-D for all reduction techniques, but no loss of mutant killing power for test suites generated by DOM-RND. This evidence suggests that STICCER's merging mechanism does not sacrifice fault-finding capability.

In terms of execution time, we find that STICCER-HGS produced comparable timings to the greedy based STICCER for reduction and the subsequent mutation analysis. Timings were marginally faster with STICCER-HGS for smaller database schemas, yet the greedy-based STICCER had the upper hand with the largest schemas, because of the additional time required by HGS to reduce suites in the first phase.

HGS and STICCER-HGS reduce test suites effectively compared to other techniques and resulting in the shortest test suites. This make the both technique more appealing for testers to inspect reduced tests and reason with fewer failed tests, lowering the human oracle cost. This however require an investigation of testers' inspection efforts either with short tests (i.e., HGS produced), or equivalent fewer tests that are long (i.e., produced by STICCER-HGS). The following section will eventuate tester's efforts to inspect differently reduced test suites using a human study.

6.4 The Human Study

6.4.1 Methodology

To investigate the effect of STICCER's test case merging mechanism on human oracle cost, a Human Study was designed in which participants acted as "testers" who had to manually inspect test suites that had been processed by STICCER. As a control, the (unmerged) test suites reduced by HGS was chosen, as they, in general, represent the smallest non-merged test suites, thereby making them suitable for the scope of a human study. As such, to allow for a direct comparison, we chose to use STICCER-HGS over the greedy-based STICCER to study the effect of test merging. A relational database test case attempts to satisfy or violate an integrity constraint with `INSERT` statements that are either accepted or rejected by the DBMS. Therefore, in this study participants had to read a test case and identify the `INSERT` statement(s) that would be rejected. We measured their accuracy and efficiency (i.e., time duration) while they performed this task, with the aim of answering two research questions:

RQ4: Test Inspection Accuracy. How accurate are humans at inspecting the merged and reduced tests produced with STICCER-HGS compared to the reduced and non-merged tests made by HGS?

RQ5: Test Inspection Duration. How long does it take for humans to inspect the merged and reduced tests produced by STICCER-HGS compared to the reduced and non-merged tests made by HGS?

Experimental Set-up

Schemas and Generators. We generated test suites using AVM-D and DOM-RND for the schemas *ArtistSimilarity*, *Inventory*, *NistXTS748*, and *Person*, as listed in Table A.1, and applied both HGS and STICCER-HGS. We deliberately picked these schemas to ensure all different types of integrity constraint were represented and a variety of data types, while also ensuring relatively small test suite sizes (i.e., under 30 test cases) so that the test suites used would be feasible for a human to inspect during the study in a reasonable amount of time.

Therefore, this study includes four schemas, two test data generators, and two reduction techniques.

Table 6.5: The Relational Database Schemas For The Human Study

			Integrity Constraints					
Schema	Tables	Columns	Check	Foreign Key	Not Null	Primary Key	Unique	Total
ArtistSimilarity	2	3	0	2	0	1	0	3
Inventory	1	4	0	0	0	1	1	2
NistXTS748	1	3	1	0	1	0	1	3
Person	1	5	1	0	5	1	0	7
Total	5	15	2	2	6	3	2	15

Test Suites were generated using *SchemaAnalyst* on SQLite, and using the ClauseAICC+ANCC+AUCC coverage criterion combination with the *mu-*

tated versions of each schema. In the study, we asked participants to assess these test suites with respect to the original schemas. We used mutants rather than original schemas for test suite generation to introduce a degree of randomness in the accept/violate pattern of the `INSERT` statements of each suite, enabling a fairer comparison between their merged and reduced versions. We randomly selected the mutant schemas summarized in Table 6.6 from a pool of mutants generated using the operators of Wright et al. [136].

Table 6.6: Selected Mutated Schemas

Schema	AVM-D	DOM-RND
ArtistSimilarity	Added a `NOT NULL` to a new column	Added a `NOT NULL` to a new column
Inventory	Changed the column of a `UNIQUE`	Changed the column of a `UNIQUE`
NistXTS748	Added a new `UNIQUE` to a new column	Added a single-column primary key
Person	Removed primary key	Changed primary key to another column

The Web-Based Questionnaire. To measure the accuracy and duration of human inspection, we integrated both the original schema and the mutant's tests into a web questionnaire. Each test case forms an individual "question", where participants are asked to select the `INSERT` statements in each test that the DBMS would reject. If the participant believed that none of the `INSERT`s should be rejected, they could select an option entitled "None of them". If a participant could not decide, then they could select the "I don't know" option. Our thinking behind both options was to prevent random guessing that could negatively influence the results. Furthermore, to prevent confounding results, we also added a mechanism that deselects checkboxes if an option was selected that would contradict another option. For instance, if a participant selected a series of `INSERT`s and then continued to pick either "I don't know" or "None of them" (i.e., they seemingly changed their mind), then the `INSERT`s are deselected, or vice versa.

At the end of questionnaire, participants were presented with an online exit survey that asked about the schemas that they thought to be the easiest and hardest to inspect. The participants could also provide general feedback regarding the questionnaire, ultimately helping us to analyze the results and further characterize a human's perception of the database schemas and their reduced test suites.

The Human Study Procedure. This study recruited 31 participants to answer this section's research questions. The participants are from the student body at the University of Sheffield, studying Computer Science (or a related degree) at either the undergraduate or PhD level. As part of the recruitment and sign-up process, potential participants completed an assessment in which they had to say whether four `INSERT` statements would be rejected for a table with three constraints. Participates were not invited if they got more than one answer wrong, ensuring that the human study included capable participants with adequate SQL knowledge.

Both Table 6.8 and Table 6.7 shows the participants' demographics. Showing SQL experience are varied between less than a year for four participants to over four years for eight. Even with all participants gaining such experience through academia, it was shown that it influenced programmers performance positively according to Diestes et al. [164]. Note four of the participants were removed from this study. The first two removed participants answered the questions wrongly, with accepted `INSERT`s rather than rejected statements. The other two removed participants used the same participant code, which we assume was a typo. Therefore, the recorded answers only showed one participant with no background question answered while the other missed answering a schema because it was already submitted.

Table 6.7: Participants Eduction Levels

Level	Participants
Postgraduate - Masters	1
Postgraduate - PhD	8
Undergraduate - Year 1	1
Undergraduate - Year 2	3
Undergraduate - Year 3	8
Undergraduate - Year 4	6

Table 6.8: Participants SQL Knowledge Length

Years	Participants
≤ 1	4
2	7
3	8
4	4
5+	4

The study had two within-subject variables (i.e., the database schemas and the generation techniques) and one between-subject variable (i.e., the specific reduced test suites), as shown in Figure 6.2. We assigned participants randomly to one of four groups, so that there were at least six participants in each group. Each group inspected each schema with each test suite, reduced by either HGS or STICCER-HGS. Each participant was financially compensated with £5 cash and £10 book voucher, encouraging them to do their best to understand the schema tests and complete the questionnaire in under an hour.

	AVM-D		DOM-RND	
Group	HGS	STICCER-HGS	HGS	STICCER-HGS
1	Schema 1	Schema 3	Schema 2	Schema 4
2	Schema 2	Schema 4	Schema 1	Schema 3
3	Schema 3	Schema 1	Schema 4	Schema 2
4	Schema 4	Schema 2	Schema 3	Schema 1

Figure 6.2: Selected Mutated Schemas

To answer RQ4, we calculated participants' test inspection accuracy scores based on the number of failing `INSERT` statements correctly selected over all the `INSERT`s (i.e., those that the DBMS accepted or rejected). We report the accuracy score's descriptive statistics (i.e., minimums, maximums, means, and medians).

To answer RQ5, we reported the same descriptive statistics for the duration of time that a human took to inspect each test suite.

Unfortunately, due to the small sample of participants and database schemas, we cannot reliably apply statistical significance tests. We leave this as an item for future work.

Threats to Validity of the Human Study

External Validity. The threat of selected schemas and its generated tests may provide results that are not evident for real schemas. This was mitigated by randomly selected four schemas that include common integrity constraints and data types in SQL schemas [22]. The latter was addressed using an open-source tool to generate tests, *SchemaAnalyst* [14], with the effective and recommend adequacy criterion [7]. This guaranteed that tests exercise all the integrity constraints as *true* and *false*. The selection of few relational schemas and tests were intentionally limited to ensure participants could complete the questionnaire in a reasonable time, also mitigating the potential negative effects of fatigue. Since no previous human studies compared reduced test suites and their human oracle cost, this study can be considered the first that is a small-scale with few participants. This results in fewer data points and did not yield statistical significant difference. Therefore, the following sections only rely upon descriptive statistics with low confidence in this chapter's claims, and in future we recommend replicating this study with larger data points to ensure there is statistical power.

Internal Validity. Participants can become better at answering questions as the questionnaire progresses which a potential learning effects and threat to internal validity. This was mitigated with randomizing the presentation order for questions and schemas. The majority of participants are students which can be considered another threat, however this can acceptable and in broad confirmation of prior results in software engineering [158].

Construct Validity. Measuring the tests understanding is subjective and a threat to the construct validity that was addressed by determining how successful human testers were at identifying which `INSERT` statement are rejected by the database violating an integrity constraint. Another threat is that the participants might not be accustomed to the questionnaire interface to determine the outcome of a SQL test case. Thus, this was addressed with a simple tutorial prior to the actual questionnaire, showing concepts of testing integrity constraints and the study's procedure. It is also possible that testers might have better knowledge of a database schema that they designed than the participants. Therefore, participants were able to study the schema understand it before showing the schema's test suite (i.e., the questions).

Table 6.9: Descriptive Statistics of Scores and Durations

The higher scores the better. Lower duration times are considered better.

						Score (%)				Duration (Minutes)			
Schema	Generator	Reduction	Participants	Tests	`INSERT`s	Min	Mean	Median	Max	Min	Mean	Median	Max
ArtistSimilarity	AVM-D	HGS	7	10	22	70.0	87.4	90.0	100.0	1.6	4.0	3.2	6.5
		STICCER-HGS	6	8	20	79.0	88.0	91.0	91.0	2.6	4.9	4.0	8.0
	DOM-RND	HGS	7	10	22	80.0	91.0	90.0	100.0	3.7	5.7	4.5	10.3
		STICCER-HGS	7	8	20	37.0	79.5	87.0	100.0	1.9	4.1	3.6	6.5
Inventory	AVM-D	HGS	7	10	16	70.0	88.6	85.0	100.0	2.7	5.7	5.4	9.6
		STICCER-HGS	7	5	12	40.0	73.4	72.0	93.0	2.1	3.2	2.7	5.7
	DOM-RND	HGS	7	7	12	78.0	94.6	100.0	100.0	1.7	3.4	3.5	5.8
		STICCER-HGS	6	6	12	25.0	83.3	100.0	100.0	1.6	2.6	2.1	5.1
NistXTS748	AVM-D	HGS	6	9	14	66.0	86.7	88.5	100.0	1.1	3.0	3.3	4.6
		STICCER-HGS	7	5	11	58.0	86.9	95.0	100.0	1.8	4.1	4.6	6.0
	DOM-RND	HGS	7	8	12	37.0	83.7	87.0	100.0	1.5	2.4	1.9	4.7
		STICCER-HGS	7	6	12	75.0	84.1	83.0	91.0	2.2	4.6	3.7	9.3
Person	AVM-D	HGS	7	11	13	31.0	85.9	90.0	100.0	1.8	4.7	3.5	10.6
		STICCER-HGS	7	3	12	88.0	95.7	100.0	100.0	1.9	4.1	2.7	9.1
	DOM-RND	HGS	6	14	19	53.0	90.8	100.0	100.0	3.8	4.9	5.0	6.3
		STICCER-HGS	7	6	18	33.0	88.3	97.0	100.0	2.8	4.3	4.0	6.5
All Schemas	AVM-D	HGS	27	40	55	31.0	87.1	90.0	100.0	1.2	4.4	4.2	10.6
		STICCER-HGS	27	21	55	40.0	85.9	91.0	100.0	1.8	4.1	3.3	9.1
	DOM-RND	HGS	27	39	65	37.0	90.0	95.0	100.0	1.5	4.1	4.0	10.3
		STICCER-HGS	27	26	62	25.0	83.8	91.0	100.0	1.6	3.9	3.5	9.3

6.4.2 Answering RQ4 and RQ5

RQ4: Test Inspection Accuracy

Table 6.9 shows the descriptive statistics for the accuracy scores of and time duration by the participants for each test suite that they evaluated. On average, participants were more accurate with the test suites reduced by HGS compared to STICCER-HGS. The mean difference in accuracy, however, for test suites was only as large as 15.2% (for the *Inventory* schema with the test suite generated by AVM-D), with the largest median difference as 13.0% (again for *Inventory* with the test suite generated by AVM-D). Overall, no clear pattern emerges, and it would seem that the smaller test suites that were reduced and merged by STICCER-HGS do not give it an advantage over HGS. This suggests that testers prefer smaller, focused test cases as much, if not more than, fewer but potentially more complex test cases.

In conclusion for RQ4, the smaller test suites reduced and merged by STICCER-HGS give it no clear advantage over test suites reduced by HGS only, suggesting that fewer, but longer, tests do not necessarily improve the accuracy of humans when they inspect test cases.

RQ5: Test Inspection Duration

Table 6.9 shows the duration descriptive statistics of each test suite inspection. For 10 of the 16 schema-test generator combinations, the participants were faster with test suites reduced and merged with STICCER-HGS, as opposed to simply being reduced with HGS. This table also shows that that the overall mean and median averages favor STICCER-HGS. These results suggest that participants can process the smaller number of test cases offered by STICCER-HGS more quickly, on the whole, even if they cannot do it more accurately. Given that STICCER-HGS test cases are longer, due to the merging, it would seem that there is more opportunity for participants to make mistakes, and/or become over-confident in their analysis.

To help understand the inspection speed of tests suites using STICCER-HGS, the exit survey used that indicate some schemas were easier to inspect based on their properties (i.e., data types).

In conclusion for RQ5, the evidence suggests that, compared to durations with tests from HGS, participants were faster at inspecting the smaller test suites reduced and merged by STICCER-HGS.

Overall Conclusions of the Human Study. The results from this study suggest that, while human testers are not more accurate at analysing a smaller number of longer tests, there is some evidence that they are faster. One explanation for this is that a tester may subconsciously spend the same amount of time on a test, regardless of its length, therefore being faster overall with smaller test suites. Yet, this constant amount of time is a disadvantage for comparatively longer tests, as there is more to inspect, and as such aspects of these test cases may be overlooked, leading to mistakes. Although interesting, these results suggest the need for a large-scale study.

6.5 Summary

Since Chapter 5 work proposed STICCER, a hybrid method that combined greedy test suite reduction with a merging approach for database schema testing, this chapter presents both a computational and a human study investigating a new hybridisation that combines STICCER-based merging with test suite reduction by the Harrold-Gupta-Soffa (HGS) method.

Considering four test suite reduction methods (i.e., greedy, HGS, STICCER, and STICCER-HGS), two test data generators (i.e., AVM-D and DOMINO), and two database schemas, this chapter's Computational Study answered three research questions. Focused on assessing the capability of these reduction methods to quickly decrease a test suite's size while preserving its mutation adequacy, the Computational Study reveals that, while there are benefits to using either greedy or HGS in combination with STICCER, neither greedy-based STICCER nor STICCER-HGS are a strictly dominant method. That is, although there was prior evidence showing that HGS was superior to greedy at reducing database schema test suites, the surprising conclusion of this study is that there is no significant benefit to hybridising STICCER with HGS instead of greedy.

Incorporating 27 participants who had to manually inspect reduced test suites and answer questions about their behaviour, the Human Study investigated the influence that STICCER's test case merging mechanism has on human oracle costs. Since this chapter's focus is on the benefits attributable to HGS, this study compared HGS to STICCER-HGS, answering two research questions. This chapter's Human Study reveals that, compared to those produced by HGS, the reduced test suites of STICCER-HGS may help humans to perform test inspection faster, but not always more accurately.

Overall, the STICCER technique is recommended for fault-finding efficiency and effectiveness. The technique also enables testers to efficiently inspect the generated test suites (i.e., evaluating the behaviour of the tests and its schema). However, testers must know that this technique can hinder the accuracy of inspecting test cases.

Chapter 7

Conclusion and Future Work

This thesis aimed to answer a high-level research question on what are the strategies that efficiently generate cost-effective database schema tests? That is to investigate and improve test data generators regarding effectiveness, efficiency, and fault-finding capabilities. Also, improving the test data generators to support testers with ease of inspection (i.e., lowering the human oracle cost) and reducing such tests for efficiency. Therefore, the thesis main objectives are as follows:

- To empirically evaluate the effectiveness and efficiency of a domain-specific test data generator against the state-of-the-art search-based technique (Chapter 3).
- To perform a human study to find understandability factors of automatically generated SQL tests (Chapter 4).
- To empirically evaluate and improve traditional test suite reduction methods in the context of database schema testing (Chapter 5).
- To perform a human study to identify testers' inspection efforts with differently reduced test suites (Chapter 6).

7.1 Summary of Contributions

The thesis achieved the mentioned aims and objectives with addressing several challenges and answering the following research questions:

1. How do domain-specific operators improve test data generators effectiveness and fault detection? How efficient are these operators at utilising the random technique for generating test data compared to the state-of-the-art search-based techniques?
2. How can different automatically generated test data influence the understanding of test cases and their expected behaviour? Moreover, which factors of these different test data are helpful for testers?
3. How do different test suite reduction techniques improve the efficiency of running mutation analysis while maintaining fault-finding capabilities? How do the merging of automatically generated test cases improve efficiency of running mutation analysis?
4. How effective do testers inspect differently reduced SQL test suites?

Therefore, the following are a summarisation of each chapter answering these research questions.

Chapter 3: "DOMINO: A Fast and Effective Test Data Generator"

This chapter presented a technique that uses domain-specific operators and random search to generate test data, called DOMINO (i.e., referred to as DOM-RND because it used random values). The empirical evaluation showed that DOM-RND was significantly efficient at generating test data and had equal coverage to AVM-D (i.e., the state-of-the-art search-based technique that uses default values). The DOM-RND also generated more diverse test data because of random values and significantly effective at finding faults compared to AVM-D. However, comparing the fault-finding capabilities of DOM-RND and AVM-D is unfair because of different initialised values. Thus, an AVM variant that uses random value, called AVM-R, was also compared to DOM-RND showing nearly equal fault detection between DOM-RND and AVM-R. Although the two techniques detected different faults because of the different values generated were selected with different methods. This investigation leads to implementing a hybridisation technique to generate suited test data for `CHECK` constraints, combining DOM-RND and AVM-R. However, results showed that the hybrid technique was less efficient at generating test data and no more effective at detecting faults.

Chapter 4: "What Factors Make SQL Test Cases Understandable For Testers?"

This chapter provided results of a conducted human study with two AVM variants, three DOMINO variants, and two schemas. The human study was conducted with two groups. A silent study group that provides quantitative results and a think-aloud group that provides qualitative results. The variants test data generators produce data as follows: (1) random values; (2) default values that use empty strings for characters and constants for numeric values; (3) values from a language model used by Afshan et al. [69], combined with a search-based technique, Alternating Variable Method (AVM); and (4) reused values derived from either column names or a library of readable values. Therefore, the techniques are AVM-D, AVM-LM (Language Model), DOM-RND, DOM-READ (uses a library of readable values), and DOM-COL (it uses column names). The human study revealed factors that influence test data understandability, such as `NULLs` are confusing, negative numbers require harder thinking, simple repetition for unimportant test values help testers, and testers prefer human-readable strings. Considering all of these factors can improve the qualitative human oracle cost of generated tests.

Chapter 5: "STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases"

This chapter presented a novel approach to test suite reduction called "STICCER", which stands for "**S**chema **T**est **I**ntegrity **C**onstraints **C**ombination for **E**fficient **R**eduction". The technique discards redundant tests using a Greedy algorithm while also merging them. This technique was able to provide test suites with decreased database interactions and restarts, resulted in faster test suite executions and mutation analysis. The empirical evaluation with 34 database schemas and compared with three general-purpose reduction technique (i.e., Random, Greedy, and HGS) showed that STICCER significantly outperformed all the techniques and the original test suite. That is, STICCER produced test suite that ran 5X and 2.5X faster than the original test suite and the reduced test suite of other techniques, respectively.

Chapter 6: "Can Human Testers Effectively Inspect Reduced Versions of Automatically Generated SQL Test Suites?"

The first part of this chapter empirically evaluated a variant STICCER technique that merges tests of HGS test suites, called STICCER-HGS. The results showed that STICCER-HGS show no significant benefit when compared to the original STICCER. Although prior evidence showing that HGS was superior to greedy at reducing database schema test suites.

The second part of this chapter conducted a human study that reviewed which of the reduced test suites, comparing HGS and STICCER-HGS, effectively enable testers to inspect tests regarding accuracy and efficiency. The human study had 27 participants, four schemas, and two test data generators (AVM-D and DOM-RND). The results revealed that the reduced test suites of STICCER-HGS, compared to those produced by HGS, may help humans to perform test inspection faster, but not always more accurately.

7.2 Limitations

This thesis includes some limitations in its empirical studies. For instance, the third chapter only compared the Domino technique to one search-based technique (i.e., AVM). However, there many other search-based techniques and constraint solvers to be compared with Domino. Therefore, the claims of the effectiveness and efficiency of Domino are only superior to AVM.

The use of using mutation analysis as a proxy to faults is also another limitation. In the absence of real faults or the history of faults, one might use mutation analysis, although mutants might not represent real-world faults and would not be detected as mutants.

The human studies presented include a limitation that it recruited a low number of participants and did not yield statistical significance, relying only on aggregated results. Therefore, it resulted in low confidence of our claims. Another limitation of the human studies is relying on small schemas as subjects, and while they might include most integrity constraints, they might not represent larger real-world schemas. Therefore, for future work, we recommend the use of larger and many schemas with more participants to yield reliable statistical significance.

7.3 Future Work

This section presents several recommendations for future work.

7.3.1 NoSQL (Non-Relational Database) Testing

With the rise of non-relational (NoSQL) databases in software development [2], I recommend devising software testing methods for such databases. The data generators with their reduction techniques and understandability factors can be applied into NoSQL. However, there might be properties that need to be observed for understandability compared to these found in this thesis.

7.3.2 Test Data Generation

Given the efficiency and effectiveness of DOM-RND, I recommend to experimentally compare it to methods that leverage constraint solvers [165]. These techniques are showing some promise to be efficient and effective in finding optimal solutions such as Microsoft Z3 SMT Solver. Furthermore, the use of methods such as evolutionary algorithms [166], and other hybrid approaches [167] should be evaluated. For traditional programs, I recommend the investigation and implementation of domain-specific operators test data generators. That is, utilising a random technique while learning from the inefficiencies of search-based methods. Another future work is to develop an automated method that quickly generates focused and effective tests for a wide variety of data-driven programs, like those that use relational databases or NoSQL data stores.

7.3.3 Test Comprehension

The findings of Chapter 4 guidelines could be used for developing new test data generators for database schemas and, when appropriate, traditional programs. The goal is to develop tools that automatically generate tests containing data values that are both understandable to humans and effective at finding faults.

The human studies conducted in this thesis should also be conducted for traditional programs and identify the understandability factors that influence test comprehension. Moreover, the human studies should be replicated with many participants and maybe in a controlled fashion to either strengthen this thesis claims or refute them.

I would also recommend the use of visualisation techniques for test suites to be more comprehended. For instance, test cases can have a commented ASCII generated tables of the `INSERT`s, which might allow testers to navigate the test data better and efficiently rather than reading `INSERT` statements. That also would require a human study to evaluate its influence on test comprehension.

7.3.4 Test Suite Reduction

Given these promising results STICCER, I recommend enhancing this method so that it operates in a multi-objective fashion, explicitly balancing testing goals like decreasing the test suite size while maximising its mutation score. Also, it provides a path toward implementing multi-objective evolutionary algorithms as an extension of the first future work.

Since STICCER has proven effective at reducing database schema test suites, I recommend that future research investigate ways in which this can be adapted to the reduction of the test suites for traditional programs that manipulate complex state in other formats. This would improve the efficiency regression testing for programs and might lower the flakiness of tests.

Also, I would recommend that conducting more human studies should be prevalent in the software testing community, such as using human studies to formulate the costs of the oracle problem. For instance, estimating the costs and benefits of using test data generators in a software development environment or continuous testing.

7.4 General Conclusions

Relational databases are critical for software systems. Testing the database schema that defines integrity constraints is crucial to ensure the consistency and coherency of storing data. Since manual schema testing is labour-intensive and error-prone, automated techniques enable the generation of test data. Although these generators are well-established, they require to be practical for testers to use. Therefore, the first contribution of this thesis evaluated the DOMINO technique (optimised random generator) against well-established methods (e.g., AVM) empirically. The second contribution was identifying understandability factors of schema tests using a human study. Thirdly, this thesis proposed and evaluated a novel approach that reduces and merge tests against traditional reduction methods. Finally, the thesis studies testers' inspection efforts with differently reduced tests using a human study. Overall, this thesis work provided an effective and efficient test data generator that can be configured for understandability and reduced for performance and ease of inspection. Therefore, helping practitioners to adopt automated test data generators in practice.

Appendices

Appendix A

Schema Subjects Table and Sources

The set of subject schemas in Table A.1 are drawn from a range of sources. *ArtistSimilarity* and *ArtistTerm* are schemas that underpin part of the Million Song dataset, a freely available research dataset of song metadata [168]. *Cloc* is a schema for the database used in the popular open-source application for counting lines of program code. While it contains no integrity constraints, test requirements are still generated since the coverage criterion I used incorporates the ANCC and AUCC criteria, discussed in Section 2.3.5. *IsoFlav_R2* belongs to a plant compound database from the U.S. Department of Agriculture, while *iTrust* is a large schema that was designed as part of a patient records medical application to teach students about software testing methods, having previously featured in a mutation analysis experiment with Java code [169]. *JWhoisServer* is used in an open-source implementation of a server for the WHOIS protocol (`http://jwhoisserver.net`). *MozillaExtensions* and *MozillaPermissions* are part of the SQLite databases underpinning the Mozilla Firefox browser. *RiskIt* is a database schema that forms part of a system for modelling the risk of insuring an individual (`http://sourceforge.net/projects/riskitinsurance`), while *StackOverflow* is the underlying schema used by the popular programming question and answer website. *UnixUsage* is from an application for monitoring and recording the use of Unix commands and *WordNet* is the database schema used in a graph visualiser for the WordNet lexical database. Other subjects were taken from the SQL Conformance Test Suite (i.e., the six "Nist–" schemas), or samples for the PostgreSQL DBMS (i.e., *DellStore*, *FrenchTowns*, *Iso3166*, and *Usda*, available from the `PgFoundry.org` website). The remainder were extracted from papers, textbooks, assignments, and online tutorials in which they were provided as examples (e.g., *BankAccount*, *BookTown*, *CoffeeOrders*, *CustomerOrder*, *Person*, and *Products*).

Table A.1: The 34 Relational Database Schemas Studied

			Integrity Constraints					
Schema	**Tables**	**Columns**	Check	Foreign Key	Not Null	Primary Key	Unique	Total
ArtistSimilarity	2	3	0	2	0	1	0	3
ArtistTerm	5	7	0	4	0	3	0	7
BankAccount	2	9	0	1	5	2	0	8
BookTown	22	67	2	0	15	11	0	28
BrowserCookies	2	13	2	1	4	2	1	10
Cloc	2	10	0	0	0	0	0	0
CoffeeOrders	5	20	0	4	10	5	0	19
CustomerOrder	7	32	1	7	27	7	0	42
DellStore	8	52	0	0	39	0	0	39
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	9	24
Inventory	1	4	0	0	0	1	1	2
Iso3166	1	3	0	0	2	1	0	3
IsoFlav_R2	6	40	0	0	0	0	5	5
iTrust	42	309	8	1	88	37	0	134
JWhoisServer	6	49	0	0	44	6	0	50
MozillaExtensions	6	51	0	0	0	2	5	7
MozillaPermissions	1	8	0	0	0	1	0	1
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	1	5	2	0	13
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	57	0	10	15	11	0	36
StackOverflow	4	43	0	0	5	0	0	5
StudentResidence	2	6	3	1	2	2	0	8
UnixUsage	8	32	0	7	10	7	0	24
Usda	10	67	0	0	31	0	0	31
WordNet	8	29	0	0	22	8	1	31
Total	186	1044	38	49	357	122	24	590

Appendix B

Replication – Running Experiments and Performing Data Analysis

This appendix explains how to run test generation experiments and data analysis with *SchemaAnalyst* and its data analysis package written in the R language for statistical computation. It will help others to use *SchemaAnalyst*, replicating the experiments of this thesis.

B.1 Chapter 3 Experiments

Since it is implemented in the Java language, *SchemaAnalyst* is cross platform. It uses the Gradle tool to manage its building, testing, and dependencies. Testers can follow the instructions at the tool's GitHub repository[1] and a previous tool paper [14] to learn how to install and run *SchemaAnalyst*. As these resources do not show how to experimentally evaluate *SchemaAnalyst*, this paper explains how to run experiments using a provided Python script called `runExperiments.py`. Because *SchemaAnalyst*'s search-based test generation methods are stochastic, testers can parameterize this script with the number of trials and a random seed in addition to giving the name of a test data generator, DBMS, and the schema under test. These are the steps for configuring and running the experiments:

1. Install *SchemaAnalyst* and one or more DBMSs.
2. Edit the `config/database.properties` file so that it provides the access details for each of the DBMSs.
3. Run the Gradle compile command, `./gradlew compile`, to install all of *SchemaAnalyst*'s dependencies.
4. Set the `CLASSPATH` to point to the tool's build directory.

[1]https://github.com/schemaanalyst/schemaanalyst

5. Modify `scripts/runExperiments.py` to configure the experiment (e.g., specify the number of trials).

An experimenter now runs the Python script, performing mutation analysis on tests generated by *SchemaAnalyst*, thereby generating the results files. Located in the `results/` directory, these files include:

1. `mutationanalysis.dat` with basic test generation and mutation information for each run;
2. `mutanttiming.dat` with details for each schema mutant both killed and alive;
3. `alive_mutant/` a directory with files and directories furnishing details about each run of data generation and mutation analysis, with notes about every live mutant.

Analysing Results. The R package[2] is provided to replicate the paper's data and tables [17]. Researchers can use `devtools` [170] to download and install the replication package and then take these steps:

1. Load the empirical results from prior experiments with:
   `mutants <- dominoR::read_analysis()`
   `analysis <- dominoR::read_mutants()`
2. To re-generate the tables in our main paper [17], a researcher can run the R package's functions (e.g.,`dominoR::table_generator_coverage`), following the provided instructions for details about inputs and outputs.
3. While the default format of the result tables is like that of our main paper, researchers can modify the replication package's code to customize table output as needed.
4. To support the generation of tables with different entries, the results analysis functions can be parameterized to, for instance, compute either mean or median values.

[2]https://github.com/schemaanalyst/domino-replicate

B.2 Chapter 4 Experiments

Django Survey Web Application[3]**.** This web application enables you to customise questionnaires depending on your requirements. This It includes two small applications that are used for Chapter 4 and 6 human studies, `polls` and `controlled`, respectively. In contains code highlighter and different types of questions (e.g., multiple choices, drop-downs, and textboxes). Participant groups (e.g., control and treatment) or any other design. The application is dependent on the following requirements:

- Python 2.7
- pip
- virtualenv

Installation Instructions. All the dependencies are wrapped using `virtualenv` and written into 'requiremnts.txt'. Therefore, two directories are required, one for dependencies and the other for the application. Follow the below instructions using Linux commands to run the application:

1. `mkdir survey`
2. `cd survey`
3. Create and activate the virtual environment:
   (a) `virtualenv envi`
   (b) `source envi/bin/activate`
4. Clone the repo:
   (a) `mkdir djsurvey`
   (b) `cd djsurvey`
   (c) `git clone https://github.com/aalshrif90/djsurvey .`
5. Install dependencies and run the web application (Run these in the 'djsurvey' directory):
   (a) `pip install -r requirements.txt`
   (b) `python manage.py runserver`

In the browser, type this URL '127.0.0.1:8000' to start with your experiment.

[3]https://github.com/aalshrif90/djsurvey

B.3 Chapter 5 and 6 Experiments

Similar to 'Chapter 3 Experiments' Section, all the reduction techniques are provided and integrated in the *SchemaAnalyst* repository. The `SampleReductionExperimentRunner.sh` script is also provided within the main root directory of the repository. Following the *SchemaAnalyst* installations, this script will run the mutation analysis 30 times for each reduction technique and the non-reduction for only three schemas. If you want to run the other schemas, please configure the script to run these for you.

The output of the experiment (i.e., mutation analysis) will be located in the 'results' directory and can be used for your analysis.

Analysing Results. The R scripts in replication package package[4] will help you replicate the paper [161] analysis and tables. To generate the tables in the paper execute the `R/tables.R` script (i.e., `source(''R/tables.R'')` in the R session). This will output latex tables in the `texTables` directory and the merges plots in the `plots` directory. To obtain the data frames for further analysis, in the R session, execute the `R/main.R` script (i.e., `source(''R/main.R'')` in the R session) and you will the `mutationanalysis` data frame.

If you have new data that was generated by *SchemaAnalyst* reduction techniques, copy the generated `results` directory to the R project root directory. Then, in the `R/main.R` change the `results_path` to point to the `results/` directory. Re-run the `R/tables.R` script to generate the tables and plots.

B.4 Summary

To conclude, this explained both how to easily run experiments with *SchemaAnalyst* and to perform data analysis with an R package and scripts. This supports the reproduction of prior experimental results and guides future researchers who want to conduct their own analyses of schema testing methods. We invite practitioners and researchers to use the test generators and mutation analysis methods provided by *SchemaAnalyst*.

[4]https://github.com/schemaanalyst/sticcer-replicate

Bibliography

[1] "Why is it important to use patient data?." https://understandingpatientdata.org.uk/why.

[2] "DB-Engines DBMS ranking." https://db-engines.com/en/ranking. 2016.

[3] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*. McGraw-Hill New York, 6th ed., 2010.

[4] K. Kravtsov, "Corruption in System Object Possibly Due to Schema or Catalog Inconsistency," Mar 2018.

[5] SQLite Developers, "Most Widely Deployed SQL Database Engine." http://www.sqlite.org/mostdeployed.html.

[6] "PostgreSQL." https://www.postgresql.org/.

[7] P. McMinn, C. J. Wright, and G. M. Kapfhammer, "The Effectiveness of Test Coverage Criteria for Relational Database Schema Integrity Constraints," *Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 8:1–8:49, 2015.

[8] E. Pollack, "The Benefits, Costs, and Documentation of Database Constraints." https://www.sqlshack.com/the-benefits-costs-and-documentation-of-database-constraints/, Dec 2019.

[9] K. Cagle, "Seven Data Modeling Mistakes." https://goo.gl/UywFw5, 2015.

[10] L. Davidson, “Ten Common Database Design Mistakes.” `https://goo.gl/kYtf2z`, 2007.

[11] B. Ericsson, “What’s Wrong With Foreign Keys?.” `https://goo.gl/zpTaYN`, 2012.

[12] J. James, “Five Common Database Development Mistakes.” `https://goo.gl/i7X1vK`, 2012.

[13] S. Guz, “Basic Mistakes in Database Testing.” `https://goo.gl/AE5haq`, 2011.

[14] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, “SchemaAnalyst: Search-based Test Data Generation for Relational Database Schemas,” in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016.

[15] B. Korel, “Automated Software Test Data Generation,” *Transactions on Software Engineering*, vol. 16, no. 8, 1990.

[16] G. M. Kapfhammer, P. McMinn, and C. J. Wright, “Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems,” in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pp. 31–40, IEEE, 2013.

[17] A. Alsharif, G. M. Kapfhammer, and P. McMinn, “DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas,” in *Proceedings of the 11th International Conference on Software Testing, Verification and Validation*, 2018.

[18] E. Dustin, T. Garrett, and B. Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education, 2009.

[19] M. Olan, “Unit Testing: Test Early, Test Often,” *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, 2003.

[20] M. Emmi, R. Majumdar, and K. Sen, “Dynamic Test Input Generation for Database Applications,” in *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[21] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, “Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem,” in *Proceedings of the 3rd International Workshop on Search-Based Software Testing*, 2010.

[22] D. Qiu, B. Li, and Z. Su, “An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications,” in *Proceedings of the 21st International Symposium on the Foundations of Software Engineering*, pp. 125–135, ACM, 2013.

[23] J. M. Rojas and G. Fraser, “Is Search-based Unit Test Generation Research Stuck in a Local Optimum?,” in *Proceedings of the 10th International Workshop on Search-Based Software Testing*, 2017.

[24] E. Daka, J. Campos, J. Dorn, G. Fraser, and W. Weimer, “Generating Readable Unit Tests for Guava,” in *International Symposium on Search Based Software Engineering*, pp. 235–241, Springer, 2015.

[25] M. J. Harrold, R. Gupta, and M. L. Soffa, “A Methodology For Controlling The Size of a Test Suite,” *Transactions on Software Engineering and Methodology*, vol. 2, no. 3, 1993.

[26] A. J. Offutt, J. Pan, and J. M. Voas, “Procedures For Reducing The Size of Coverage-based Test Sets,” in *Proceeding of the 12th International Conference on Testing Computer Software*, 1995.

[27] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, “Empirical studies of test-suite reduction,” *Software Testing, Verification and Reliability*, vol. 12, no. 4, 2002.

[28] P. McBreen, *Software Craftsmanship: The New Imperative.* Addison-Wesley Professional, 2002.

[29] J. Jazequel and B. Meyer, “Design by Contract: The Lessons of Ariane,” *Computer*, vol. 30, no. 1, pp. 129–130, 1997.

[30] P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge University Press, 2008.

[31] G. J. Myers and C. Sandler, *The Art of Software Testing.* John Wiley & Sons, 2004.

[32] A. Bouguettaya, M. Ouzzani, B. Medjahed, and J. Cameron, “Managing Government Databases,” *Computer*, no. 2, pp. 56–64, 2001.

[33] S. Ambler, “Database Testing: How to Regression Test a Relational Database.” `http://www.agiledata.org/essays/databaseTesting.html`. (Accessed 24/01/2014).

[34] S. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design.* Pearson Education, 2006.

[35] C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo, “Schema Evolution in Wikipedia: Toward a Web Information System Benchmark,” in *International Conference on Enterprise Information Systems (ICEIS)*, 2008.

[36] I. Sommerville, *Software Engineering.* Addison-wesley, 2011.

[37] E. W. Dijkstra, “Notes on Structured Programming,” 1970.

[38] J. C. King, “Symbolic Execution and Program Testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[39] L. A. Clarke, “A program testing system,” in *Proceedings of the 1976 annual conference*, pp. 488–491, ACM, 1976.

[40] P. P. Mahadik and D. Thakore, “Survey on Automatic Test Data Generation Tools and Techniques For Object-oriented Code,” *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 4, pp. 357–364, 2016.

[41] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[42] L. Cseppento and Z. Micskei, "Evaluating Symbolic Execution-based Test Tools," in *Proceeding of the 8th International Conference on Software Testing, Verification and Validation*, pp. 1–10, IEEE, 2015.

[43] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[44] J. W. Duran and S. Ntafos, "A Report on Random Testing," in *Proceedings of the 5th international conference on Software engineering*, pp. 179–183, IEEE Press, 1981.

[45] M. Harman, Y. Jia, and Y. Zhang, "Achievements, Open Problems and Challenges for Search Based Software Testing," in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, pp. 1–12, IEEE, 2015.

[46] W. Afzal, R. Torkar, and R. Feldt, "A Systematic Review of Search-based Testing for Non-functional System Properties," *Information and Software Technology*, vol. 51, pp. 957–976, June 2009.

[47] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An Orchestrated Survey on Automated Software Test Case Generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[48] J. Kempka, P. McMinn, and D. Sudholt, "Design and Analysis of Different Alternating Variable Searches for Search-Based Software Testing," *Theoretical Computer Science*, vol. 605, 2015.

[49] M. Harman and P. McMinn, "A Theoretical and Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Gen-

eration," in *International Symposium on Software Testing and Analysis (ISSTA 2007)*, pp. 73–83, ACM, 2007.

[50] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical Evaluation of Optimization Algorithms When Used in Goal-oriented Automated Test Data Generation Techniques," *Empirical Software Engineering*, vol. 12, no. 2, pp. 183–239, 2007.

[51] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419, ACM, 2011.

[52] P. McMinn, "IGUANA: Input Generation Using Automated Novel Algorithms. a plug and play research tool," Tech. Rep. CS-07-14, Department of Computer Science, University of Sheffield, 2007.

[53] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An Empirical Evaluation of Evolutionary Algorithms for Unit Test Suite Generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[54] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[55] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression Mutation Testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

[56] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *Transactions on Software Engineering*, vol. 41, no. 5, 2015.

[57] B. Hailpern and P. Santhanam, "Software Debugging, Testing, and Verification," *Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[58] R. Duke, G. Rose, and G. Smith, "Object-Z: A Specification Language Advocated for the Description of Standards," *Computer Standards & Interfaces*, vol. 17, no. 5-6, pp. 511–533, 1995.

[59] J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen, "The B-method," in *International Symposium of VDM Europe*, pp. 398–405, Springer, 1991.

[60] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta, "Assessing, Comparing, and Combining State Machine-based Testing and Structural Testing: A Series of Experiments," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 161–187, 2010.

[61] D. Coppit and J. M. Haddox-Schatz, "On The Use of Specification-based Assertions as Test Oracles," in *29th Annual IEEE/NASA Software Engineering Workshop*, pp. 305–314, IEEE, 2005.

[62] Y. Cheon, "Abstraction in Assertion-based Test Oracles," in *Seventh International Conference on Quality Software (QSIC 2007)*, pp. 410–414, IEEE, 2007.

[63] S. J. Prowell and J. H. Poore, "Foundations of Sequence-based Software Specification," *IEEE transactions on Software Engineering*, vol. 29, no. 5, pp. 417–429, 2003.

[64] M. D. Davis and E. J. Weyuker, "Pseudo-oracles For Non-testable Programs," in *Proceedings of the ACM'81 Conference*, pp. 254–257, ACM, 1981.

[65] R. Feldt, "Generating Diverse Software Versions with Genetic Programming: An Experimental Study," *IEE Proceedings-Software*, vol. 145, no. 6, pp. 228–236, 1998.

[66] P. McMinn, "Search-based Failure Discovery Using Testability Transformations to Generate Pseudo-oracles," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 1689–1696, ACM, 2009.

[67] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[68] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System For Dynamic Detection of Likely Invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[69] S. Afshan, P. McMinn, and M. Stevenson, "Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost," in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pp. 352–361, IEEE, 2013.

[70] P. McMinn, M. Stevenson, and M. Harman, "Reducing Qualitative Human Oracle Costs associated with Automatically Generated Test Data," in *Proceedings of the International Workshop on Software Test Output Validation*, 2010.

[71] E. Daka, J. M. Rojas, and G. Fraser, "Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?," in *Proceedings of International Symposium on Software Testing and Analysis*, 2017.

[72] R. Elmasri and S. Navathe, *Fundamentals of Database Systems.* Pearson Education India, 2008.

[73] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts.* New York, NY, USA: McGraw-Hill, Inc., 5 ed., 2006.

[74] R. Sint, S. Schaffert, S. Stroka, and R. Ferstl, "Combining Unstructured, Fully Structured and Semi-structured Information in Semantic Wikis," in *CEUR Workshop Proceedings*, vol. 464, pp. 73–87, 2009.

[75] E. F. Codd, "A Relational Model of Data For Large Shared Data Banks," *Communications of the ACM*, vol. 13, 1970.

[76] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL Faults in Database Applications," in *Proceedings of the 26th International Conference on Automated Software Engineering*, pp. 213–222, IEEE Computer Society, 2011.

[77] D. R. Slutz, "Massive Stochastic Testing of SQL," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998.

[78] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A Genetic Approach For Random Testing of Database Systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 1243–1251, VLDB Endowment, 2007.

[79] M. Poess and J. M. Stephens, Jr., "Generating Thousand Benchmark Queries in Seconds," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, vol. 30, 2004.

[80] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "QAGen: Denerating Query-aware Test Databases," in *Proceedings of the 2007 SIGMOD International Conference on Management of Data*, 2007.

[81] A. Arasu, R. Kaushik, and J. Li, "Data Generation Using Declarative Constraints," in *Proceedings of the 2011 International Conference on Management of Data*, 2011.

[82] A. Arasu, R. Kaushik, and J. Li, "DataSynth: Generating Synthetic Data using Declarative Constraints," *Proceedings of the 37th International Conference Very Large Data Bases*, vol. 4, no. 12, 2011.

[83] S. Khalek, B. Elkarablieh, Y. Laleye, and S. Khurshid, "Query-Aware Test Generation Using a Relational Constraint Solver," in *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008.

[84] S. A. Khalek and S. Khurshid, "Automated SQL Query Generation for Systematic Testing of Database Engines," in *Proceedings of the 25th International Conference on Automated Software Engineering*, 2010.

[85] N. Bruno and S. Chaudhuri, "Flexible Database Generators," in *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[86] K. Houkjær, K. Torp, and R. Wind, "Simple and Realistic Data Generation," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[87] G. M. Kapfhammer and M. L. Soffa, "A Family of Test Adequacy Criteria for Database-driven Applications," in *Proceedings of the 9th European Software Engineering Conference and the 11th Symposium on the Foundations of Software Engineering*, vol. 28, 2003.

[88] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-Based Testing of Database Application Programs with Conceptual Data Model," in *Proceedings of the 5th International Conference on Quality Software*, 2005.

[89] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA For Testing Relational Database Applications," *Software Testing, Verification and Reliability*, vol. 14, no. 1, pp. 17–44, 2004.

[90] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full Predicate Coverage for Testing SQL Database Queries," *Software Testing, Verification and Reliability*, vol. 20, no. 3, pp. 237–288, 2010.

[91] M. J. Suárez-Cabal and J. Tuya, "Improvement of Test Data by Measuring SQL Statement Coverage," in *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice*, 2003.

[92] W. G. J. Halfond and A. Orso, "Command-Form Coverage for Testing Database Applications," in *Proceedings of 21st International Conference on Automated Software Engineering*, pp. 69–80, IEEE, 2006.

[93] Y. Deng, P. Frankl, and D. Chays, "Testing Database Transactions with AGENDA," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 78–87, IEEE, 2005.

[94] F. Haftmann, D. Kossmann, and E. Lo, "A Framework for Efficient Regression Tests on Database Applications," *The VLDB Journal*, vol. 16, no. 1, 2007.

[95] A. Arcuri and J. P. Galeotti, "SQL Data Generation to Enhance Search-based System Testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1390–1398, ACM, 2019.

[96] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya, "Constraint-based Test Database Generation for SQL Queries," in *Proceedings of the 5th International Workshop on the Automation of Software Test*, pp. 67–74, ACM, 2010.

[97] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "SQLMutation: A Tool to Generate Mutants of SQL Database Queries," in *Proceedings of the 2nd Workshop on Mutation Analysis*, 2006.

[98] J. Tuya, C. de la Riva, M. J. Suarez-Cabal, and R. Blanco, "Coverage-aware Test Database Reduction," *Transactions on Software Engineering*, vol. 42, no. 10, 2016.

[99] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for SQL queries," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 1220–1230, ACM, 2018.

[100] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An Automated Framework For Structural Test-data Generation," in *Proceedings of the 13th International Conference on Automated Software Engineering*, 1998.

[101] C. J. Wright, G. M. Kapfhammer, and P. McMinn, "The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mu-

tation Analysis," in *Proceedings of the 14th International Conference on Quality Software*, pp. 57–66, IEEE Computer Society, 2014.

[102] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri, "Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?," *Software Testing, Verification and Reliability*, vol. 28, no. 4, 2018.

[103] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?," in *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation*, 2015.

[104] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing Container Classes: Random or Systematic?," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 262–277, Springer, 2011.

[105] P. McMinn, C. Wright, C. McCurdy, and G. M. Kapfhammer, "Automatic Detection and Removal of Ineffective Mutants For The Mutation Analysis of Relational Database Schemas," *Transactions on Software Engineering*, 2019.

[106] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically Documenting Unit Test Cases," in *Proceedings of the 9th International Conference on Software Testing, Verification and Validation*, pp. 341–352, IEEE, 2016.

[107] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Aiding Comprehension of Unit Test Cases and Test Suites with Stereotype-Based Tagging," in *Proceedings of the 26th International Conference on Program Comprehension*, 2018.

[108] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "Documenting Database Usages and Schema Constraints in Database-Centric Applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[109] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing Test Suites to Aid in Software Understanding," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007.

[110] A. M. Smith, J. J. Geiger, G. M. Kapfhammer, M. Renieris, and G. E. Marai, "Interactive Coverage Effectiveness Multiplots For Evaluating Prioritized Regression Test Suites," in *Compendium of the 15th Information Visualization Conference*, 2009.

[111] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated Unit Test Generation During Software Development: A Controlled Experiment and Think-aloud Observations," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[112] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An Empirical Investigation on The Readability of Manual and Generated Test Cases," in *Proctedings of the 26th International Conference on Program Comprehension*, 2018.

[113] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving Code Readability Models with Textual Features," in *Proceeding of the 24th International Conference on Program Comprehension*, pp. 1–10, IEEE, 2016.

[114] A. Ahadi, J. Prior, V. Behbood, and R. Lister, "Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries," in *Proceedings of the Conference on Innovation and Technology in Computer Science Education*, 2016.

[115] P. Reisner, "Human Factors Studies of Database Query Languages: A Survey and Assessment," *Computing Surveys*, vol. 13, no. 1, 1981.

[116] T. Taipalus, M. Siponen, and T. Vartiainen, "Errors and Complications in SQL Query Formulation," *Transactions on Computing Education*, vol. 18, no. 3, 2018.

[117] J. B. Smelcer, "User Errors in Database Query Composition," *International Journal of Human-Computer Studies*, vol. 42, no. 4, 1995.

[118] J. A. Jones and M. J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *Transactions on Software Engineering*, vol. 29, no. 3, 2003.

[119] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, 2012.

[120] V. Chvatal, "A Greedy Heuristic For The Set-Covering Problem," *Mathematics of Operations Research*, vol. 4, no. 3, 1979.

[121] T. Y. Chen and M. F. Lau, "A Simulation Study on Some Heuristics For Test Suite Reduction," *Information and Software Technology*, vol. 40, no. 13, 1998.

[122] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," in *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[123] H. Zhong, L. Zhang, and H. Mei, "An experimental comparison of four test suite reduction techniques," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[124] S. Tallam and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm For Test Suite Minimization," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 35–42, 2006.

[125] D. Jeffrey and N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction," *Transactions on software Engineering*, vol. 33, no. 2, 2007.

[126] C.-T. Lin, K.-W. Tang, and G. M. Kapfhammer, "Test Suite Reduction Methods That Decrease Regression Testing Costs by Identifying Irreplaceable Tests," *Information and Software Technology*, vol. 56, no. 10, 2014.

[127] C.-T. Lin, K.-W. Tang, J.-S. Wang, and G. M. Kapfhammer, "Empirically Evaluating Greedy-Based Test Suite Reduction Methods at Different Levels of Test Suite Complexity," *Science of Computer Programming*, vol. 150, 2017.

[128] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-Criteria Models For All-Uses Test Suite Reduction," in *Proceeding of the 26th International Conference on Software Engineering*, 2004.

[129] J. Hartmann and D. Robson, "Revalidation During The Software Maintenance Phase," in *Proceeding of the Conference on Software Maintenance*, 1989.

[130] N. Mansour and K. El-Fakih, "Simulated Annealing and Genetic Algorithms For Optimal Regression Testing," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 1, 1999.

[131] S. Yoo and M. Harman, "Using Hybrid Algorithm For Pareto Efficient Multi-Objective Test Suite Minimisation," *Journal of Systems and Software*, vol. 83, no. 4, 2010.

[132] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-Grained Test Minimization," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, p. 210–221, Association for Computing Machinery, 2018.

[133] S. Ambler, "Adopting Evolutionary/Agile Database Techniques." `http://www.agiledata.org/essays/adopting.html`.

[134] G. Kapfhammer, "Towards a Method for Reducing the Test Suites of Database Applications," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012.

[135] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Query-Aware Shrinking Test Databases," in *Proceedings of the Second International Workshop on Testing Database Systems*, 2009.

[136] C. J. Wright, G. M. Kapfhammer, and P. McMinn, “Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation,” in *Proceedings of the 8th International Workshop on Mutation Analysis*, 2013.

[137] R. A. DeMillo and A. J. Offutt, “Constraint-Based Automatic Test Data Generation,” *Transactions on Software Engineering*, vol. 17, no. 9, 1991.

[138] R. DeMillo, E. Krauser, and A. Mathur, “Compiler-Integrated Program Mutation,” in *Proceedings of the 15th Annual International Computer Software and Applications Conference*, 1991.

[139] G. M. Kapfhammer, “Software Testing,” in *The Computer Science Handbook*, CRC Press, 2004.

[140] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, “Mutating Database Queries,” *Information and Software Technology*, vol. 49, no. 4, pp. 398–417, 2007.

[141] S. Poulding and J. A. Clark, “Efficient Software Verification: Statistical Testing Using Automated Search,” *Transactions on Software Engineering*, vol. 36, no. 6, 2010.

[142] A. Arcuri and L. Briand, “A Hitchhiker’s Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering,” *Software Testing, Verification and Reliability*, vol. 24, pp. 219–250, May 2014.

[143] A. Vargha and H. D. Delaney, “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong,” *Journal of Education and Behavioral Statistics*, vol. 25, no. 2, 2000.

[144] A. Alsharif, G. M. Kapfhammer, and P. McMinn, “What Factors Make SQL Test Cases Understandable For Testers? A Human Study of Automatic Test Data Generation Techniques,” in *Proceedings of the 35th International Conference on Software Maintenance and Evolution*, 2019.

[145] SchemaAnalyst Repository, "SchemaAnalyst Test Data Generation Tool." `https://github.com/schemaanalyst/schemaanalyst`.

[146] A. Gibson, "Generate Test Data with DataFactory." `https://www.andygibson.net/blog/article/generate-test-data-with-datafactory/comment-page-1/`, 2011.

[147] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, no. 4, 1987.

[148] A. Von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, no. 8, 1995.

[149] A. A. Rupp, T. Ferne, and H. Choi, "How Assessing Reading Comprehension with Multiple-Choice Questions Shapes The Construct: A Cognitive Processing Perspective," *Language Testing*, vol. 23, no. 4, 2006.

[150] Y. Ozuru, S. Briner, C. A. Kurby, and D. S. McNamara, "Comparing Comprehension Measured by Multiple-Choice and Open-Ended Questions," *Canadian Journal of Experimental Psychology*, vol. 67, no. 3, 2013.

[151] D. R. Bacon, "Assessing Learning Outcomes: A Comparison of Multiple-Choice and Short-Answer Questions in a Marketing Context," *Journal of Marketing Education*, vol. 25, no. 1, 2003.

[152] PostgreSQL, "SQL Conformance." `https://www.postgresql.org/docs/9.5/static/features.html`.

[153] DigitalOcean, "SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems." `https://goo.gl/mrZSG4`, July 2017.

[154] J. W. Creswell, R. Shope, V. L. Plano Clark, and D. O. Green, "How Interpretive Qualitative Research Extends Mixed Methods Research," *Research in the Schools*, vol. 13, no. 1, 2006.

[155] G. Charness, U. Gneezy, and M. A. Kuhn, "Experimental Methods: Between-Subject and Within-Subject Design," *Journal of Economic Behavior & Organization*, vol. 81, no. 1, 2012.

[156] J. Nielsen, T. Clemmensen, and C. Yssing, "Getting Access to What Goes on in People's Heads? Reflections on The Think-Aloud Technique," in *Proceedings of the 2nd Nordic Conference on Human-Computer Interaction*, 2002.

[157] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg, *The Think Aloud Method: A Practical Approach to Modelling Cognitive*. Academic Press, 1994.

[158] M. Höst, B. Regnell, and C. Wohlin, "Using Students as Subjects: A Comparative Study of Students and Professionals in Lead-Time Impact Assessment," *Empirical Software Engineering*, vol. 5, no. 3, 2000.

[159] Y. Xia, J. Sun, and D.-G. Chen, "Power and Sample Size Calculations for Microbiome Data," in *Statistical Analysis of Microbiome Data with R*, Springer, 2018.

[160] ANSI/ISO/IEC International Standard – ISO/IEC 9075-2:2011, "Database Language SQL — Part 2: Foundation (SQL/Foundation)," 2011.

[161] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "STICCER: Fast and Effective Database Test Suite Reduction Through Merging of Similar Test Cases," in *International Conference on Software Testing, Verification and Validation (ICST 2020) (To Appear)*, 2020.

[162] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency," in *Proceedings of the 24th International Symposium on Software Testing and Analysis*, 2015.

[163] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "Hybrid methods for reducing database schema test suites: Experimental insights from com-

putational and human studies," in *Proceedings of the 1st IEEE/ACM International Conference on Automation of Software Test*, 2020.

[164] O. Dieste, A. M. Aranda, F. Uyaguari, B. Turhan, A. Tosun, D. Fucci, M. Oivo, and N. Juristo, "Empirical Evaluation of The Effects of Experience on Code Quality And Programmer Productivity: An Exploratory Study," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2457–2542, 2017.

[165] S. Khalek and S. Khurshid, "Systematic Testing of Database Engines Using a Relational Constraint Solver," in *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, 2011.

[166] P. Tonella, "Evolutionary Testing of Classes," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2004.

[167] P. McMinn and M. Holcombe, "Evolutionary Testing Using an Extended Chaining Approach," *Evolutionary Computation*, vol. 14, no. 1, 2006.

[168] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The Million Song Dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval*, 2011.

[169] B. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," in *Proceedings of the 3rd International Workshop on Mutation Analysis*, 2007.

[170] "Devtools." `https://github.com/hadley/devtools`.