Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Test Data Generation for Relational Database Applications

David Chays

Department of Computer and Information Science

Technical Report
TR-CIS-2005-01
01/12/2005

TEST DATA GENERATION FOR RELATIONAL DATABASE APPLICATIONS

DISSERTATION

Submitted in Partial Fulfillment

of the Requirements for the

Degree of

DOCTOR OF PHILOSOPHY

(Computer & Information Science)

at the

POLYTECHNIC UNIVERSITY

by

D. Chays

January 2004

Approved :

_____

Department Head

Copy No. _____

_____

Approved by the Guidance Committee :

Major : Computer & Information Science

Phyllis Frankl

Professor of
Computer & Information Science

Gleb Naumovich

Assistant Professor of
Computer & Information Science

Filippos Vokolos

Assistant Professor of
Computer & Information Science

Minor : Electrical Engineering

Shivendra Panwar

Professor of
Electrical Engineering

VITA

David Chays was born in Brooklyn, New York in November 1972. He received his B.S. degree in Computer Science from Brooklyn College of the City University of New York in 1995 and his M.S. degree in Computer Science from Polytechnic University in Brooklyn, New York, in 1998. After passing the Ph.D. Qualifying Exam at Polytechnic University in 1999, he began working on the database application testing project leading to his thesis, under the supervision of Phyllis Frankl. His research interests are in the areas of software testing, database systems, and computer security.

The work presented in this thesis was supported by teaching fellowships from the department of Computer and Information Science at Polytechnic University and grants from the National Science Foundation and the Department of Education.

ACKNOWLEDGMENTS

My sincerest gratitude to my advisor, Professor Phyllis Frankl, who gave me the opportunity to pursue my academic goals and whose extraordinary support and guidance helped me to reach those goals. Her expertise and insight kept me focused in the right direction. She also encouraged me to pursue areas that interested me, even those that were not directly related to my thesis, such as the anti-virus project, which was a wonderful learning experience. I greatly appreciate that she took an interest in my overall professional growth and well-being. I am very fortunate to have had such a caring and knowledgeable advisor.

My work is part of a comprehensive database application testing project. I would like to give special thanks to Phyllis Frankl, Elaine Weyuker (AT&T Research) and Filippos Vokolos (Assistant Professor at Drexel University), who were instrumental in the development of the ideas which led to this thesis as well as the ongoing project. I am also grateful to my committee members: Gleb Naumovich, Filippos Vokolos, and Shivendra Panwar, for their support and their help in improving this dissertation. I appreciate very much the support and encouragement I have received from the chairman of the department of Computer and Information Science, Stuart Steele, as well as all the faculty in the department.

I would like to thank my colleagues and friends. From the time I joined Polytechnic University, Vinay Kanitkar and Ravanasamudram Uma were always there for me when I needed help or advice. I learned so much from Uma and Vinay; we had many interesting and insightful conversations. I had the pleasure and privilege of working with Saikat Dan on two projects. Dan is a team player and also a great friend. Dan continued to contribute ideas and assistance long after he was no longer officially on the project, including help with some technical difficulties which inhibited progress on the project. I would also like to thank Zhongchiang Chen, Yuetang Deng, and James Wang for their contributions to the database application testing project. It was great sharing an office with Uma and then with Allen Chang; it's truly nice to work alongside such great people. I will never forget some of the conversations with Je-Ho Park, and the good times we shared as friends. I met so many good people here at Poly in addition to those already mentioned, whom I would like to thank for their friendship and collegiality and with whom I would like to stay in touch: Holden Anele, Yu Chen, Tom Cortina, Angelo Curreli, Mark Hoffman, Jimmy Ip, Shawn Li, Saurabh Majumdar, Yuval Marton, Zan Ouyang, Roy Rajarshi, Al Rotondaro, Granit Sharon and Ray Valdez. A special thanks to Robert Siegfried, my Compilers teacher, who is now my colleague at Adelphi University, and Yu Chen who was my colleague at Poly and is presently my colleague at Adelphi. Mark Hoffman, Robert Siegfried and Joel Wein showed an enthusiasm for teaching that helped inspire me to teach; I am fortunate to have had such great teachers, not just here at Poly, but also at Brooklyn College where I received my Bachelors degree: Michael Barnett, Chaya Gurwitz, Seth Schacher, Gerald Weiss. There are many people who had a positive influence on me and I am thankful to them all.

Friends whom I met outside of Poly were there for me for the good times and the bad: Qi Chen, Keven Friedman, Henry Fuerte, Seth Schacher, and Harry and Sally Shpelfogel.

Steve, my Jiu-Jitsu instructor, helped me focus at a crucial time, by explicitly showing me the dramatic improvement in results achieved just by focusing.

My accomplishments would have been neither possible nor meaningful without the love and support of my parents, Helene and Seymour, my brother Marty, and my fiancée Lauren Faivus. When I had doubts about my ability to finish, I thought about how happy my family and friends would be for me if I completed my Ph.D. and that was enough motivation for me to persist until I finished what I started.

AN ABSTRACT

TEST DATA GENERATION FOR RELATIONAL DATABASE APPLICATIONS

by

David Chays
Advisor: Phyllis Frankl

Submitted in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy (Computer Science)

January 2004

Database systems play an important role in nearly every modern organization, yet relatively little research effort has focused on how to test them. This dissertation discusses issues arising in testing database systems, presents an approach to testing database applications, and describes AGENDA, a set of tools to facilitate the use of this approach. In testing such applications, the state of the database before and after the user's operation plays an important role, along with the user's input and the system output. A framework for testing database applications is introduced. A complete tool set, based on this framework, has been prototyped. The components of this system are: a parsing tool that gathers relevant information from the database schema and application, a tool that populates the database with meaningful data that satisfy database constraints, a tool that generates test cases for the application, a tool that checks the resulting database state after operations are performed by a database application, and a tool that assists the tester in checking the database application's output. This dissertation focuses on the tools involved in generating the database state and inputs for the application. A case study based on the TPC-C benchmark shows promising results.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Databases (DBs) play a central role in the operations of almost every modern organization. Commercially-available database management systems (DBMSs) provide organizations with efficient access to large amounts of data, while both protecting the integrity of the data and relieving the user of the need to understand the low-level details of the storage and retrieval mechanisms. To exploit this widely-used technology, an organization will often purchase an off-the-shelf DBMS, and then design database schemas and application programs to fit its particular business needs.

It is essential that these database systems function correctly and provide acceptable performance. Substantial effort has been devoted to insuring that the algorithms and data structures used by DBMSs work efficiently and protect the integrity of the data. However, relatively little attention has been given to developing systematic techniques for assuring the correctness of the related application programs. Given the critical role these systems play in modern society, there is clearly a need for new approaches to assessing the quality of the database application programs. To address this need, we developed a systematic, partially-automatable approach to testing database applications and a tool set based on this approach.

There are many aspects to the correctness of a database system, including the following:

1. Does the application program behave as specified?

2. Does the database schema correctly reflect the organization of the real world data being modeled?

3. Are security and privacy protected appropriately?

4. Are the data in the database accurate?

5. Does the DBMS perform all insertions, deletions, and updates of the data correctly?

All of these aspects of database system correctness, along with various aspects of system performance, are vitally important to the organizations that depend on the database system. This thesis focuses on the first of these aspects of correctness, the correctness of database application programs.

Many testing techniques have been developed to help assure that programs meet their specifications, but most of these have been targeted toward programs written in traditional imperative languages. We believe that new approaches targeted specifically toward testing database applications are needed for several reasons. A database application program can be viewed as an attempt to implement a function, just like programs developed using traditional paradigms. However, considered in this way, the input and output spaces include the database states as well as the explicit input and output parameters of the application. This has substantial impact on the notion of what a test case is, how to generate test cases, and how to check the results produced by running the test cases. Furthermore, DB application programs are usually written in a semi-declarative language, such as SQL, or a combination of an imperative language and a declarative language, such as a C program with embedded SQL, rather than using a purely imperative language. Most existing program-based software testing techniques are designed explicitly for imperative languages, and therefore are not directly applicable to the DB application programs of interest here.

The important role of database systems, along with the fact that existing testing tools do not fit well with the nature of database applications, imply that effective, easy-to-use testing techniques and tool support for them are real necessities for many organizations. This thesis discusses issues that arise in testing database applications, describes an approach to testing such systems, and describes a tool set based on this approach. We restrict attention to relational databases. Section 1.3 discusses the most closely-related commercial tools and research papers. Section 1.2 reviews relevant background and terminology. Chapter 2 discusses issues arising in testing database applications, describes our approach, and presents an overview of the tool set we built, called AGENDA, A (test) GENerator for Database Applications. Chapters 3 and 4 discuss the roles of integrity constraints, combinatorial issues, and heuristics in guiding AGENDA's test data generation. Chapter 5 discusses AGENDA's operation on the

TPC-C benchmark schema and application. Chapter 6 concludes with a summary of our contributions and directions for future work.

1.2 Background and Terminology

1.2.1 Relational Databases and SQL

Relational databases are based on the relational data model, which views the data as a collection of relations [11, 12, 21]. Relations are often thought of as tables in which each row represents data about a particular entity and each column represents a particular aspect of that data. A *relation schema* $R(A_1, \ldots, A_n)$ is a relation name (table name) along with a list of attributes (column names), each of which has a name A_i and a domain (type) $dom(A_i)$. The domains must be atomic types, such as integers or strings, rather than more complex types, such as records. A *relation* or *relation state* of the relation schema R is a set of tuples, each of which is an element of the Cartesian product $dom(A_1) \times \ldots \times dom(A_n)$. A relation schema describes the structure of the data, while a relation describes the state of the data at a particular moment in time. The relation schema is fixed at the time the database is designed and changes very infrequently, whereas the relation state is constantly modified to reflect changes in the real world entity that is being modeled. In particular, the relation state changes when there is a new entry in the database or when an entry is deleted or modified.

A *relational database schema* is a set of relation schemas along with a set of integrity constraints. The integrity constraints restrict the possible values of the database states so as to more accurately reflect the real-world entity that is being modeled. A *relational database state* is a set of relation states of the given relations, such that the integrity constraints are satisfied.

There are several types of constraints:

1. *Domain constraints* specify the possible values of an attribute; they may, for example, restrict integer values to a given sub-range.

2. *Uniqueness constraints* specify that no two distinct tuples can have the same values for a specified collection of attributes. If the number of attributes involved is more than one, then this is referred to as a *composite constraint*.

3. Ordinarily, in addition to values in dom(A), attribute A can take on a special value called NULL. A *not-NULL constraint* specifies that a given attribute cannot take on the NULL value.

4. *Referential integrity constraints* (also known as *foreign key constraints*) state that values of a particular attribute in one table R_1 must also appear as values of a particular attribute in another table R_2. For example, let R_1 be a table listing all of a company's employees and their associated department number, and let R_2 be a table listing all departments and their heads. Then if Employee X is a member of Department Y, there will be an entry indicating that in R_1, and there must also be an entry for Department Y in table R_2.

5. *Semantic integrity constraints* are general constraints on the values of a database state, expressed in some constraint specification language. For example, these may express business rules of the organization whose data is being modeled, such as a requirement that the department head's salary should be higher than that of any other member of the department.

SQL is a standardized language for defining and manipulating relational databases [14].[1] SQL includes a *data definition language (DDL)* for describing database schemata, including integrity constraints, and a *data manipulation language (DML)* for retrieving information from and updating the database. It also includes mechanisms for specifying and enforcing security constraints, for enforcing integrity constraints, and for embedding statements into other high-level programming languages. SQL allows the user to express operations to query and modify the database in a very high-level manner, expressing what should be done, rather than how it should be done. SQL and dialects thereof are widely used in commercial database systems, including *Oracle*^{TM} and *MSAccess*^{TM}.

A *trigger* is a stored procedure that executes or fires under specialized circumstances, such as when a specified table has had rows inserted, deleted, or updated [21]. The validation components of AGENDA use triggers as the means of capturing changes in tables.

A *transaction* is a collection of operations that perform a single logical function in a database application [34]. By providing the properties of atomicity, isolation, and durability, the DBMS ensures that concurrent execution of consistent transactions preserves the relationship between the state of the database and the state of the enterprise being modeled despite failures. Producing correct and consis-

[1] We will use SQL to refer to the 1992 standard, also known as SQL2 or SQL-92, and dialects thereof, unless otherwise noted.

tent transactions is the sole responsibility of the application programmer [30]. Transactions are studied further in Chapter 5. This thesis focuses on testing the correctness of database applications. Testing database transaction concurrency is discussed in [18].

Figure 1.1a provides an example of a database schema in SQL representing a database with two tables. Tables `dept` and `emp` hold data about departments and employees who work in those departments. Constraints indicate the primary key for each table (a kind of uniqueness constraint). The primary key constraint for table `dept` is defined on attribute `deptno`; this indicates that no two rows can have the same entry in this column. Similarly, the primary key constraint for table `emp` is defined on attribute `empno`. The referential integrity constraint "`foreign key(deptno) references dept`" indicates that each department number (`deptno`) appearing in table `emp` must appear in table `dept`. The check constraint "`check((salary` $\geq$ `6000.00) and (salary` $\leq$ `10000.00))`" indicates that all values for the attribute `salary` must be between 6000.00 and 10000.00 inclusive.

The database application program which we would like to test consists of code written in a high-level language, such as C, with SQL queries embbeded in the program. Embedded SQL refers to SQL statements placed within an application program. A parameterized query is a SQL query with *host variables*, which are variables declared in the host program; within the SQL statements, host variables are preceded by colons, as in Figure 1.1b. In this thesis, we initially focus on transactions which consist of a single parameterized query. Chapter 5 discusses AGENDA's operation on the TPC-C benchmark schema and transactions consisting of multiple queries.

Two examples of embedded SQL queries are the `select` and `update` queries in Figure 1.1b. A host variable in an SQL query represents either an input parameter or an output parameter. Host variables in the WHERE clause of any query or the SET clause of an update query, such as `rate`, `in_empno` and `in_deptno`, are input parameters. Host variables in the INTO clause, such as `out_name` and `out_bonus`, are output parameters. Generating a test case involves instantiating each input parameter with an appropriate value. Validating a test case involves examining the output of a `select` query and/or examining the resulting DB state in order to check that it changed or did not change appropriately. A `select` statement reads from the database and therefore does not modify the DB state, whereas an `update`, `insert` or `delete` statement potentially modifies the DB state. In general, an SQL query can retrieve many tuples. Typically, the host language program processes the retrieved tuples one at a time via a *cursor*. A cursor can be thought of as a pointer that points to a single tuple (row) from the result of a query [21].

a) A database schema definition in SQL

```
CREATE TABLE dept( deptno INT, dname CHAR(20), loc CHAR(20),
PRIMARY KEY(deptno) );

CREATE TABLE emp( empno INT PRIMARY KEY, ename CHAR(25) UNIQUE NOT NULL,
salary MONEY, bonus MONEY, deptno INT, FOREIGN KEY(deptno) REFERENCES
dept, CHECK( (salary ≥ 6000.00) AND (salary ≤ 10000.00) ) );
```

b) Example queries

```
UPDATE emp SET salary = salary * :rate WHERE ( (emp.empno = :in_empno)
AND (salary ≥ 5000.00 AND salary ≤ 10000.00) );

SELECT ename, bonus INTO :out_name, :out_bonus FROM emp WHERE
( (emp.deptno = :in_deptno) AND (salary > 7000.00 AND salary ≤ 9000.00) );
```

Figure 1.1: Examples: a) database schema definition and b) queries

Our goal is to assist the database application developer or tester in a usable, useful way, in the selection of consistent and comprehensive data values for the DB state and input test cases. Data is consistent if it does not violate the integrity constraints intended and specified by the tester. Data is comprehensive if it includes many different situations in order to increase the likelihood of exposing faults in the application program. Our approach leverages the fact that the database schema is described formally in the Data Definition Language of SQL, in order to ensure that the data we generate satisfies the integrity constraints specified in the schema, and allows the user to provide additional information to guide the generation. We want to automate the testing process as much as possible, but not at the expense of usability. We don't want to burden users with having to describe their data and/or applications in yet another language, as is required by some other approaches.

1.3 Related Work

There is little work in the software testing research literature on testing techniques targeted specifically toward database applications. Davies, Beynon, and Jones [15] populate a database with a prototype that requires the user to provide validation rules to specify constraints on attributes. Tsai, Volovik, and Keefe [38] automatically generate test cases from relational algebra queries. There are also some practical guidelines for practitioners, as in [2].

Several techniques and tools have been proposed for automated or partially automated test generation for imperative programs. Most of these attempt the difficult task of identifying constraints on the input that cause a particular program feature in an imperative program to be exercised and then use heuristics to try to solve the constraint [16, 28, 24, 27]. Zhang, Xu, and Cheung [43] generate a set of constraints which collectively represent a property against which the program is tested.

The approach of Chan and Cheung [4] is to transform the embedded SQL statements into procedures in some general-purpose programming language, and thereby generate test cases using conventional white box testing techniques. This approach suggests an alternative implementation of our input generation tool, but is not as complete as our approach since it does not consider the role of the database state and the integrity constraints as described in the schema, in generating test cases, nor does it allow the tester to provide additional information to guide the generation.

Our technique is more closely related to techniques used for specification-based test generation, e.g. [39]. The partially automated Category-Partition technique [33], introduced by Ostrand and Balcer, and described in Section 4.1, is close in spirit to our proposed test generation technique. Dalal et al. introduced another closely-related requirements-based automated test generation approach and tool, which they call model-based testing [13]. Unlike Category-Partition testing or model-based testing, however, the document that drives our technique is not a specification or requirements document but rather, a formal description of part of the input (and output) space for the application.

Chen et al [10] have developed a framework that supports Category-Partition test case generation. Each valid combination of the input parameters' choices or groups corresponds to a test frame, or template for test cases. The tester guides the generation of test frames by specifying relationships between different input parameters. These relationships are checked for consistency and automatically deduced when feasible. The tester also has the option to specify relative priorities for different groups, thus reducing the amount of test data generated. However, issues which are relevant for testing database applications, such as generation of data that are consistent with integrity constraints (in particular, uniqueness, referential integrity and composite key constraints) and interactions among database state generation, input generation and validation of the resulting state and output, are not considered.

There are similarities between database systems and object-oriented systems, where the state of an object is important when testing a method [22, 20, 9, 29]. A good survey on testing object-oriented software can be found in [1]. However, there are important differences discussed in Section 2.1.

The database literature includes some work on testing database systems, but it is generally aimed at assessing the performance of database management systems, rather than testing applications for correctness. Several benchmarks have been developed for performance testing DBMS systems [37, 3]. Another aspect of performance testing is addressed by Slutz [35], who has developed a tool to automatically generate large numbers of SQL DML statements with the aim of measuring how efficiently a DBMS (or other SQL language processor) handles them. In addition, he compares the outputs of running the SQL statements on different vendors' database management systems in order to test those systems for correctness.

Gray et al. [25] have considered how to populate a table for testing a database system's performance, but their goal is to generate a huge table filled with dummy data having certain statistical properties. In contrast, we are far less interested, in general, in the quantity of data in each of the tables. Our primary interest is rather to assure that we have reasonably "real" looking data for all of the tables in the DB, representing all of the important characteristics that have been identified and which, in combination with appropriate user inputs, will test a wide variety of different situations the application could face.

Of all previous work that we have identified as having any relevance to ours, perhaps the closest is an early paper by Lyons [31]. This work is motivated by similar considerations, and the system reads a description of the structure of the data and uses it to generate tuples. Lyons developed a special purpose language for the problem. An approach similar to that of Lyons is taken by the commercial tool DB-Fill (http://www.bossi.com/dbfill). To use DB-Fill, the tester must produce a definition file describing the schema in a special purpose language. Another commercial tool, DataFactory (http://www.quest.com/datafactory), provides the tester with options to insert existing values, sequential values, or random values, as well as an option to choose null probability. Like our approach, Lyons, DB-Fill and DataFactory rely on the user to supply possible values for attributes, but they do not handle integrity constraints nearly as completely as our approach, nor do they provide the tester with the opportunity to partition the attribute values into different data groups. By using the existing schema definition, our approach relieves the tester of the burden of describing the data in yet another language and allows integrity constraints to be incorporated in a clean way. In addition, our technique provides support for the checking of the resulting database state and output; the related approaches do not provide support for this important aspect of testing.

Chapter 2

Overview of AGENDA architecture and high level description of issues

In this chapter, we discuss several issues that arise in testing database applications and motivate the architecture of the AGENDA tool set. We illustrate these issues with the following simple hypothetical example, an application program that a company might use to process a request for a salary increase for an employee.

Assume the database includes a department table, with information about departments, including department number, department name, and department location, and an employee table, with employees' ID numbers, names, salaries, bonuses, and the departments for which they work, as in Figure 1.1a. The application program's specification is as follows:

> Input an employee's ID number and the amount of salary increase requested. If the ID number or the amount is invalid, return code 0; otherwise, if the employee works in a location in which that amount of increase is allowed, and the employee has earned a bonus of at least the amount requested, add the requested amount to the employee's salary, update the employee table appropriately, and send a notice to the payroll department; return code 1. If the employee does not work in a location in which the amount requested is allowed, return code 2. If the amount requested is allowed, but the employee is not qualified for the increase due to insufficient bonus, return code 3.

2.1 The Role of the Database State

A database application, like any other program, can be viewed as computing a (partial) function from an input space *I* to an output space *O*. Its specification can be expressed as a function (or, more generally, a relation) from *I* to *O*. Thus, we can test a database application by selecting values from *I*, executing the application on them, and checking whether the resulting values from *O* agree with the specification. However, unlike "traditional" programs considered in most of the testing literature and by most existing testing tools, the input and output spaces have a complicated structure, which makes selecting test cases and checking results more difficult.

On the surface, it appears that the inputs to the program are the employee ID and the amount of salary increase requested, while the output is a numeric code between 0 and 3. This suggests that generating test cases is a matter of finding various "interesting" employee IDs and salary increments, and that checking each result involves examining a single integer. However, a moment's reflection reveals that the expected and actual results of executing the application on a given (employee-ID, salary increment) pair also depend on the state of the database before executing the application. Similarly, knowledge of expected and actual values of the database state after executing the application are needed in order to determine whether the application behaved correctly. For example, the intended behavior of the program on a given (employee-ID, salary increment) pair depends on whether the employee ID is valid, what bonus the employee has earned, and where the employee works. In addition to the employee table and the department table, a table containing information about the maximum salary increment allowed per location will also have to be accessed, and is therefore germane to the behavior of the application.

There are several possible approaches to dealing with the role of the database state:

1. Ignore the database state, viewing the application as a relation between the user's inputs and user's outputs. This is obviously unsuitable, since such a mapping would be non-deterministic, making it essentially impossible to validate test results or to re-execute test cases.

2. Consider the database state as an aspect of the environment of the application and explicitly consider the impact of the environment in determining the expected and actual results of the user's input.

3. Treat the database state as part of both the input and output spaces.

If the environment is modeled in sufficient detail in approach (2), then approaches (2) and (3) are equivalent. In these approaches, the well-known problems of *controllability* and *observability* arise. Controllability deals with putting a system into the desired state before execution of a test case and observability deals with observing its state after execution of the test cases.

These problems have been studied in the context of communications protocols, object-oriented systems and other systems whose behavior on a particular input is dependent on the system state [9, 19, 20, 29, 23]. Several techniques for testing switching circuits, as well as techniques derived from these algorithms and targeted toward protocol testing or more general state-based systems, use an explicit finite-state or Markovian model of the system under test to achieve controllability and observability.

Unlike such systems, the set of database states cannot be easily characterized by a finite state model. The state space is well-structured, as described by the schema, but it is essentially infinite and it is difficult to describe transitions between states concisely. In testing database applications, the controllability problem manifests itself as the need to populate the database with appropriate data before executing the application on the user's input so that the test case has the desired characteristic. For example, increasing an employee's salary would give rise to several different test cases including:

- The employee ID is invalid.
- The salary increment is invalid.
- The employee qualifies for the increase and the increase is allowed in the location where the employee works.
- The employee qualifies for the increase, but the increase is not allowed in the location where the employee works.
- The increase is allowed in the location where the employee works, but the employee does not qualify for the increase.
- The employee does not qualify for the increase and the increase is not allowed in the location where the employee works.

Each of these cases represents information in the database for the employee for whom a salary increase is requested. Since the status of the employee (as stored in the database state) is different in

each of these cases, the expected output would also be different in each of these cases, even for the same input value (employeeID, increment). The observability problem manifests itself as the need to check the state of the database after test execution to make sure the specified modifications, and no others, have occurred.

2.2 Selecting Interesting Database States

As demonstrated above, the expected results of a given test case depend not only on the employee ID and the increment selected, but also on the state of the database, including whether the employee has earned the increase by having a sufficient bonus, and whether or not the increment is allowed in the location in which the employee works. To save effort, it might be possible to create one original DB state representing several different employees, having a variety of different characteristics.

Populating the DB with meaningful values may involve the interplay between several tables. For example, in Figure 1.1a, selecting values for attribute `deptno` of table `emp` depends on the values already selected for attribute `deptno` of table `dept`, due to the referential integrity constraint. More complicated interactions involve composite key constraints. A table may have a composite primary key and/or one or more composite foreign keys, and an attribute may be involved in both a composite primary key and a composite foreign key; there are many such instances in the TPC-C benchmark application, presented in Chapter 5. Handling of composite keys is discussed in Chapter 3.

Interactions can be even more complicated. For example, there might be a table $T1$ specifying the maximum increment allowed in each location. Assume that each row in $T1$ is a pair of (location, maximum increment). To create a good initial DB state for the above situation, one would need to include a record for an employee who currently has a bonus X and works for a department in location Y where (Y, Z) is in $T1$, X is greater than or equal to the requested increment, and Z is less than the requested increment, indicating that the employee has a sufficient bonus to qualify for the salary increase, but the amount exceeds that which is allowed in the area where the employee works. When the employee's manager tries to increase the employee's salary in the above situation, the application program should deny permission, since, although the employee qualifies by virtue of his/her bonus, the amount of increase is not allowed due to a restriction imposed by the location in which the employee works. Even more complicated rules indicating permissible combinations, as well as complex rules describing inter-

actions, could exist. For example, there could be a rule that no employee in a given department can earn more than the manager of that department.

2.3 Observing the Database State After Test Execution

Similarly, in checking the results, the tester must pay attention to the state of the database after running the application: If everything went smoothly, has the employee been added to the list of those employees who had salary changes? Has an appropriate change been made in the employee table? If the employee was not qualified for the increase, or if the salary increase was not allowed, has the database state been left unchanged?

Note that there is some interplay between the DB states before and after running the application and the inputs supplied by the user. Suppose entries, indicating that employee 12345 works in a location for which the maximum salary increase allowed is 99.99, have been included in the initial DB state in order to test the situation in which the request cannot be granted. This should be communicated to the tester, indicating that the pair (12345, 100.00) should be included as an input, and that the resulting output code 2 with no associated change to the DB state should be expected.

2.4 Populate DB with Live or Synthetic Data?

One approach to obtaining a DB state is to simply use live data (the data that are actually in the database at the time the application is to be tested). This involves building an exact replica of a database and performing testing on this “real” data, albeit in the laboratory. Otherwise, if testing is done with data currently in use, then any changes made to the database state would have to be undone to avoid corrupting the DB, and this may be difficult, particularly if the changes are extensive or if other “real” modifications to the DB are being performed concurrently with testing.

However, even if it is feasible to create an exact copy of the live data, there are several disadvantages to this approach. The live data may not reflect a sufficiently wide variety of possible situations that could occur. That is, testing would be limited to, at best, situations that could occur given the current DB state. Even if the live data encompasses a rich variety of interesting situations, it might be difficult to find them, especially in a large DB, and difficult to identify appropriate user inputs to exercise them

and to determine appropriate outputs. Finally, there may be security or privacy constraints that prevent the application tester from seeing the live data.

2.5 Generating synthetic data

Since using live data is problematic, our approach is to generate data specifically for the purpose of testing and to run the tests in an isolated environment. Since the DB state is a collection of relation states, each of which is a subset of the Cartesian product of some domains, it may appear that all that is needed is a way of generating values from the given domains and "gluing" them together to make tables. However, this ignores an important aspect of the database schema: the integrity constraints. We do not want to populate the DB with just any tables, but rather, with tables that contain both valid and interesting data. If the DBMS enforces integrity constraints, one could attempt to fill it with arbitrary data, letting it reject data that doesn't satisfy the constraints. However, this is likely to be a very inefficient process. Instead, we will try to generate data that is known to satisfy the constraints and then populate the DB with it. Furthermore, we would like to select that data in such a way as to include situations that the tester believes are likely to expose faults in the application or are likely to occur in practice, to assure that such scenarios are correctly treated. In order to ensure that the data are valid, we can take advantage of the database schema, which describes the domains, the relations, and the constraints the database designer has explicitly specified. Our approach leverages the fact that this information is expressed in a formal language, SQL's Data Definition Language (DDL). This makes it possible to automate much of the process.

2.6 System Overview

To address the issues described above, we have developed a tool set, AGENDA, to help test database applications. The AGENDA architecture is shown in Figure 2.1. AGENDA takes as input the database schema of the database on which the application runs; the application source code; and "sample-values files", containing some suggested values for attributes. The user interactively selects test heuristics and provides information about expected behavior of test cases. Using this information, AGENDA populates the database, generates inputs to the application, executes the application on those inputs

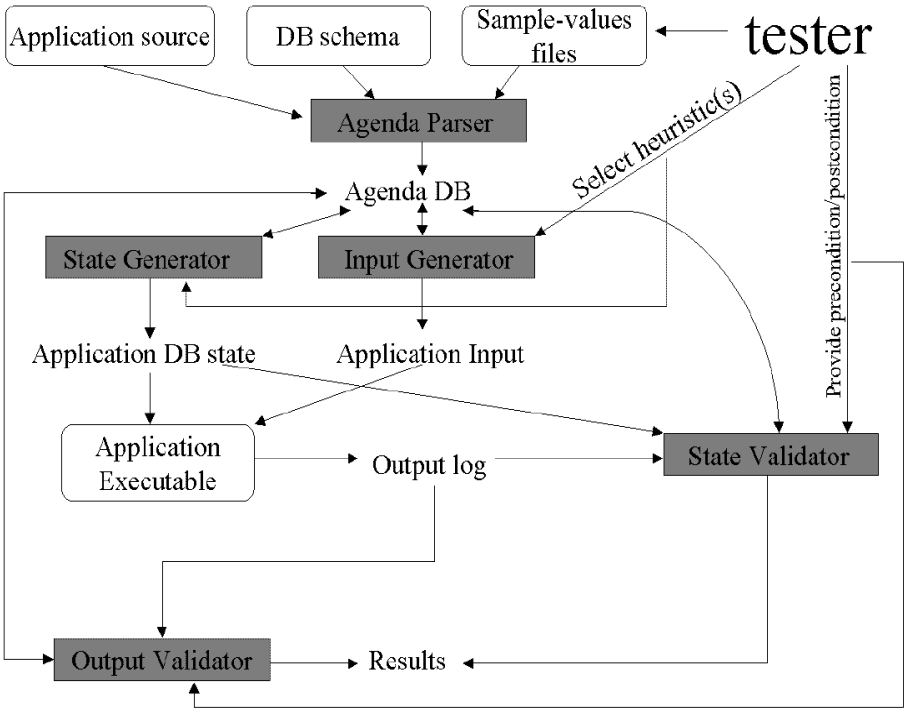


Figure 2.1: Architecture of the AGENDA tool set

and checks some aspects of correctness of the resulting database state and the application output. We initially assume that the application consists of a single SQL query. Chapter 5 relaxes this assumption.

The approach is loosely based on the Category-Partition method [33]: the user supplies suggested values for attributes, partitioned into groups, which we call *data groups*.[1] This data is provided in the sample-values files. The tool then produces meaningful combinations of these values in order to fill database tables and provide input parameters to the application program. Data groups are used to distinguish values that are expected to result in different application behavior, e.g. different categories of employees. Additional information about data groups can also be provided via annotations, as described in Section 4.2.

Using these data groups and guided by heuristics selected by the tester, AGENDA populates the DB and produces a collection of *test templates* representing abstract test cases. The tester then provides information about the expected behavior of the application on tests represented by each template. For example, the tester might specify that the application should increase the salaries of employees in the

[1] In the Category-Partition method, these data groups are called “choices”.

"faculty" group by 10% and should not change any other attributes.

In order to control the explosion in the number of test templates and to force the generation of particular kinds of templates, the tester selects heuristics. Available heuristics include one to favor "boundary values", heuristics to force the inclusion of NULL values where doing so is not precluded by not-NULL constraints, heuristics to force the inclusion of duplicate values where so doing is not precluded by uniqueness constraints, and heuristics to force the inclusion of values from all data groups.

Finally, AGENDA instantiates the templates with specific test values, executes the test cases and checks that the outputs and new database state are consistent with the expected behavior indicated by the tester.

AGENDA consists of five interacting components that operate with guidance from the tester. The first component (*Agenda Parser*) extracts relevant information from the application's database schema, the application queries, and tester-supplied sample-values files, and makes this information available to the other four components. It does this by creating an internal database, which we refer to as the *Agenda DB*, to store the extracted information. The Agenda DB is used and/or modified by the remaining four components.

The second component (*State Generator*) uses the database schema along with information from the tester's sample-values files indicating useful values for attributes (optionally partitioned into different groups of data), and populates the database tables with data satisfying the integrity constraints. It retrieves the information about the application's tables, attributes, constraints, and sample data from the Agenda DB and generates an initial DB state for the application, which we refer to as the *Application DB*. Heuristics, described in more detail later, are used to guide the generation of both the application DB state and inputs.

The third component (*Input Generator*) generates input data to be supplied to the application. The data are created by using information that is generated by the Agenda Parser and State Generator components, along with information derived from parsing the SQL statements in the application program and information that is useful for checking the test results. For example, if two rows with identical values of attribute a_i are generated for some table in order to test whether the application treats duplicates correctly, information is logged to indicate the attribute value, and this is used to suggest inputs to the tester. Information derived from parsing the application source code may also be useful in suggesting inputs that the tester should supply to the application. Using the Agenda DB, along with the tester's

choice of heuristics, the Input Generator instantiates the input parameters of the application with actual values, thus generating test inputs.

The fourth component (*State Validator*) investigates how the state of the application DB changes during execution of a test. It automatically logs the changes in the application tables and semi-automatically checks the state change.

The fifth component (*Output Validator*) captures the application's outputs and checks them against the query preconditions and postconditions that have been generated by the tool or supplied by the tester.

This thesis focuses on the parsing component (Agenda Parser) and generation components (State Generator and Input Generator). The checking components (State Validator and Output Validator) are described further in [8].

The Agenda Parser is described in the next sub-section. In Section 4.2, we use the schema and queries in Figure 1.1 to illustrate the operations of the State Generator and Input Generator. The `update` query involves a possible change in the application's DB state, so the components involved are: Agenda Parser, State Generator, Input Generator, and State Validator. The `select` query should not change the application's DB state, so the components involved are: Agenda Parser, State Generator, Input Generator, and Output Validator.

2.6.1 Agenda Parsing Tool

At the core of the Agenda Parsing tool is an SQL parser. We have chosen to base the tool on PostgreSQL, an object-relational DBMS, originally developed at UC Berkeley [36] and now commercially supported by PostgreSQL [26]. PostgreSQL supports most of SQL-2 (along with additional features that are more object-oriented) and provides a well-documented open-source parser. Given a schema, the PostgreSQL parser creates an Abstract Syntax Tree that contains relevant information about the tables, attributes, and constraints. However, this information is spread out in the tree, making it inconvenient and inefficient to access during test generation. Furthermore, it is possible to use different SQL DDL syntactic constructs to express the same underlying information about a table; consequently, the location of the relevant information in the tree is dependent on the exact syntax of the schema definition. For example, the primary key constraints on tables `dept` and `emp` in Figure 1.1a are expressed using different syntactic constructs, leading to different parse sub-tree structures.

For these reasons, rather than forcing AGENDA components to work directly with the Abstract Syn-

tax Tree, the PostgreSQL parser was modified so that as it parses the schema definition for the database underlying the application to be tested, it collects relevant information about the tables, attributes, and constraints. An earlier version of the tool, described in [5], stored this information in a complex, dynamically expanding data structure. The current version of the tool, Agenda Parser, stores this information in a database, called the Agenda DB, that is internal to our system. Thus, the memory issues associated with the original data structure, as discussed in [5], are avoided, and the information is more accessible to the other components of AGENDA, which are now database applications. In addition, this design choice has made it much easier to modify all the AGENDA components.

Some of the information that is stored in the Agenda DB is also stored in the DBMS's internal catalog tables. In an alternative design, the remaining components, the State Generator, Input Generator, State Validator, and Output Validator, could query these catalog tables. However, building and then querying a separate Agenda DB allows us to decouple the remaining components from the details of PostgreSQL. This allows AGENDA to be ported to a different DBMS by changing only the Agenda Parser. In addition, the catalog does not contain all of the information needed by the AGENDA components, so some additional parsing would be needed even using this approach. Furthermore, as described below, the Agenda DB is also used to store additional information supplied by the tester and to store information needed for checking test results. The Agenda DB is a repository of information for the tools to read and update, and in effect, communicate with one another. It facilitates handling of the interplay between the database state, the application inputs, and the expected results. Key portions of the Agenda DB's schema are shown in Appendix A.

The PostgreSQL parser is generated by YACC from a collection of grammar rules and associated actions. The Agenda Parser was created by modifying selected actions, and traversing selected structures, in order to gather relevant data into the Agenda DB when it is identified during parsing. For example, the actions for the CREATE TABLE statement produce the *column definition* and *table constraint* nodes of the Abstract Syntax Tree, as in Figure 2.2. Note that the information about constraints can appear in a column definition node or a table constraint node, depending on the syntax used; for example, if the attribute name precedes the constraint, then this information is stored in a column definition node; if the constraint precedes the attribute name, then this information is stored in a table constraint node, as in Figure 2.2. Information about tables, attributes, and constraints, dispersed in these nodes of the Abstract Syntax Tree, are gathered and stored in the Agenda DB.

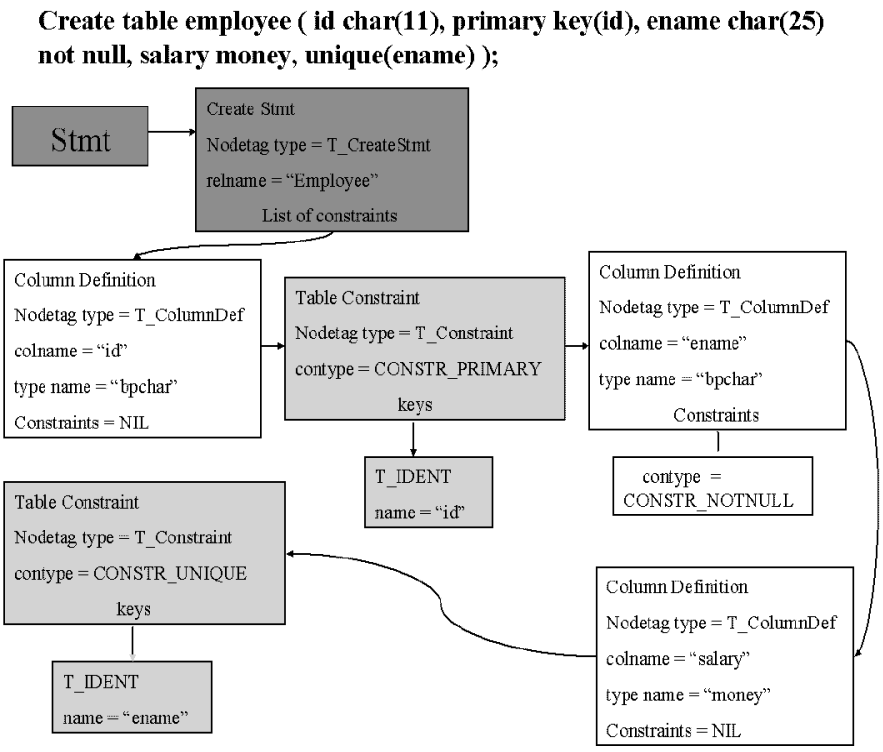


Figure 2.2: Information in the Abstract Syntax Tree for a CREATE TABLE statement

As the Agenda Parser parses the (application) schema, it extracts information about tables, attributes, and constraints from the Abstract Syntax Tree, as described above. This version of the Agenda Parser extracts information about uniqueness constraints and referential integrity constraints (including composite constraints involving multiple attributes), and not NULL constraints. It also extracts limited information from semantic constraints, namely boundary values from sufficiently simple boolean expressions. For the table `emp` in Figure 1.1a, the Agenda Parser extracts boundary values 6000.00 and 10000.00 along with their associated attributes (in this case, `salary` for both) from the Abstract Syntax Tree and stores this information in the Agenda DB.

After parsing the application schema, the Agenda Parser parses the sample-values files and stores the given sample values, their data groups, and their associated attributes in the Agenda DB. Attributes involved in composite constraints involve special handling. Attributes that are parts of composites are marked as such in the Agenda DB so that they can be handled correctly during test generation.

Next the application query is parsed, in order to extract information about the parameters to the query. This information is needed for test generation and output checking. For queries embedded in a

host program, the parser must extract information about *host variables*, variables in the host language that are used as parameters in SQL queries. Again, the DBMS parser was modified to accomplish this. A small modification to the lexical analyzer was needed in order to allow the parser to accept queries with uninstantiated host variables. Information is extracted by traversing the query tree built by the DBMS. Clauses with complicated expressions and many input and output host variables can be handled cleanly. Further details are provided in [7].

The query tree is traversed to identify input and output host variables. Input host variables are found in the `where` clause of a query and the `set` clause of an `update` query. Output host variables are found in the `into` clause of a `select` query. For example, as the `select` query in Figure 1.1b is parsed, Agenda Parser stores in the Agenda DB the following information: `out_name` and `out_bonus` are output host variables associated with attributes `ename` and `bonus` respectively, belonging to the table `emp`, and `in_deptno` is an input host variable associated with attribute `deptno` of table `emp`. For the `update` query in Figure 1.1b, Agenda Parser stores data indicating that `in_empno` is an input host variable associated with `emp.empno`, and `rate` is an input host variable associated with `emp.salary`. If there is an association between an input host variable and an attribute, the type of association is also stored in the Agenda DB. A *direct* association between an input host variable and an attribute indicates to the Input Generator that when it instantiates a value for this input host variable, it can choose a sample value among those supplied by the tester for the associated attribute. For example, `in_empno` is directly associated with `empno`, meaning that the Input Generator can choose values among those supplied for the attribute `empno` in order to instantiate `in_empno`. An *indirect association* between an input host variable and an attribute indicates to the Input Generator that when it instantiates a value for this input host variable, it cannot choose a sample value from those supplied for the associated attribute. For example, `rate` is indirectly associated with `salary`, indicating to the Input Generator that it should not choose a value among those supplied for the attribute `salary`, but should choose a value from another source, either from the application source (if possible) or from a different tester-supplied file (with sample values of rates, as opposed to salaries).

Boundary values can also be easily identified in the query tree. For the `select` query in Figure 1.1b, Agenda Parser extracts the boundary values 7000.00 and 9000.00 along with the associated attribute (`salary`) from the query tree and stores this information in the Agenda DB. There may also be boundary values defined in the schema, as discussed above. Once extracted and stored in the Agenda DB, these

table Table_recs

Table	Number of attributes	Order to fill (refer to subsection 3.2.2)
dept	3	0
emp	5	1

table Attribute_recs

Attribute	Type	Constraints
dept.deptno	int	primary key
dept.dname	char(20)	no constraints
dept.loc	char(20)	no constraints
emp.empno	int	primary key
emp.ename	char(25)	unique, not null
emp.salary	money	check constraints
emp.bonus	money	no constraints
emp.deptno	int	foreign key referencing dept.deptno

table Boundary values

Attribute	Value	Operator
emp.salary	6000.00	$\geq$
emp.salary	10000.00	$\leq$

Figure 2.3: Information in the Agenda DB after parsing the schema

boundary values are used in two ways. They are used by the Agenda Parser to automatically partition an input domain into data groups, as described in sub-section 4.1.1. The boundary values are also used by the State Generator and Input Generator for selection of values to insert into the Application DB and test cases, respectively, if the tester selects the `boundary values` heuristic to guide the generation.

2.6.2 Example

The following example shows the kind of information (meta-data) stored in the internal database, the Agenda DB, by the Agenda Parser. The Agenda DB schema is provided in Appendix A. The tools which use the information in the Agenda DB are described later.

With reference to the application schema in Figure 1.1a and the `update` query in Figure 1.1b, suppose we know that the value of host variable `:rate` always comes from the application (e.g. by doing some data flow analysis on the application source code), so we only generate test cases for host variable `:in_empno`. By parsing the query, we know it is associated with attribute `empno` in table `emp`. Suppose in the sample-values file for `empno`, the sample values provided for attribute `empno` are partitioned into 3 groups: student, faculty, and administrator, as in Figure 4.1a.

The Agenda Parser stores information in the Agenda DB for this schema, as shown in Figure 2.3.

Based on parsing the application query and sample-values files, the Agenda Parser stores information in the Agenda DB, indicating that the query has host variables `:rate`, indirectly associated with attribute `salary`, and `:in_empno` directly associated with `empno`; that it potentially changes attribute `salary` in table `emp`; that there are three data groups for attribute `emp.empno: student, faculty, administrator`. It generates three templates, `template_emp_empno_student`, `template_emp_empno_faculty`, `template_emp_empno_administrator`, and stores two boundary values for attribute `salary: 5000.0, 100000.0`.

2.6.3 Walk-through

An overview from the user's perspective involves interacting with the AGENDA system via a web browser, as follows [6]:

- The user logs into the AGENDA system.
- The user is requested to upload a tar file containing the database schema, the application with embedded SQL queries, and a collection of files (sample-values files) containing sample values for each attribute. The user may choose to partition the values into different data groups based on knowledge that the application might treat certain groups of values differently. For example, the database schema may contain a table about employees and a table about the departments for which they work; the application may update salaries of employees who meet certain criteria; the attribute corresponding to employee number (`empno`) may be partitioned into 3 data groups (student, faculty, and administrator) to reflect different categories of employees.
- The user chooses heuristics to guide the State Generator, as described above. The State Generator produces an initial database state that is consistent with the schema and sample-values files provided by the user.
- AGENDA generates test templates for the input parameters, based on the information about data groups associated with each parameter. For each test template, the user has the option to provide preconditions and postconditions.
- The user chooses heuristics to guide the Input Generator, as described above. The Input Generator produces test cases for the application by instantiating all input parameters with actual values.

Suppose the application consists of updating the salary of a particular employee whose employee number (`empno`) matches that supplied as input (`:input_empno`), and the user selects the ”all groups” heuristic. In this case, there is only one input parameter (`:input_empno`) which the Input Generator needs to instantiate; in accordance with the heuristic chosen, the Input Generator generates enough test cases so that each data group associated with `empno` is represented.

- The user selects test cases, on which AGENDA will run the application.
- AGENDA executes each test case, checks the post-condition for the corresponding template, and reports a message indicating success or failure.

Chapter 3

Integrity Constraints

The database schema designer restricts the values of certain attributes by specifying integrity constraints. Section 1.2 provides definitions and examples of integrity constraints. Assuming the DBMS enforces integrity constraints, we could generate arbitrary data and let the DBMS reject those values that violate constraints. However, if many values are rejected, this is a very inefficient process, so instead, when possible, we generate data that satisfy the integrity constraints.

Honoring integrity constraints is necessary both for the State Generator in populating the database and for the Input Generator in generating test cases. However, when the Input Generator selects values from the database, it leverages the work done by the State Generator to populate the database with data that are consistent with the integrity constraints. Unless otherwise stated, the handling of integrity constraints described in this thesis, and this section in particular, applies directly to the State Generator and indirectly to the Input Generator.

3.1 Uniqueness constraints

Agenda DB table *data_group_recs* has one row for each data group. Agenda DB table *value_recs* has one row for each value. If there is a uniqueness constraint on a single attribute, the appropriate frequency fields in the *data_group_recs* and *value_recs* tables are checked to avoid selecting the same value more than once. When a data group and value are selected for the database, the corresponding fields *choice_freq_in_db* and *val_freq_in_tc* are incremented, respectively. If all values have been used for an attribute which is unique, the State Generator cannot produce additional tuples; otherwise, the

uniqueness constraint would be violated.

3.2 Referential integrity constraints

3.2.1 Selecting appropriate tuples

Referential integrity (foreign key) constraints are handled as follows: When selecting a value for attribute A in table T, where this attribute references attribute A' in table T', the State Generator refers to the value records associated with the attribute records for attribute A' in table T' and selects a value that has already been used.

3.2.2 Filling tables in a correct order

If there is a reference from table T to table T', then table T' should be filled before table T is filled; otherwise, there will be a referential integrity violation, since the key referenced from T is not found in T'.

The current tool implementation uses a topological sort to impose an ordering on the application table names, stored in the Agenda DB, so that a table that is referenced is filled before tables that reference it, assuming there is no referential cycle. This is done by labelling each node (table) with a reference type i $\geq$ 0 such that each node it references has type j $<$ i. In the first round of labelling, all nodes that do not reference any other nodes are labelled with type 0. In the second round, all nodes that reference only nodes of type 0 are labelled with type 1. In the third round, all nodes that reference only nodes of type 0 and 1 are labelled with type 2. In general, in round i $>$ 1, all nodes that reference nodes of type less than i and at least 0 are labelled with type i-1. Assuming there is no referential cycle, all nodes are labelled with a reference type, such that if we populate tables in ascending order of their reference types, each referenced table is filled before any table referencing it.

This algorithm can be modified to handle a referential cycle. According to the SQL standard, if attribute A of table T references attribute A' of table T' then A' cannot reference A, because any insertion would violate the foreign key constraint; this would mean that column A in T is an exact duplicate of the corresponding column A' in T'; if the user really wanted that, he/she would have to use another means, such as a trigger (as defined in Section 1.2). However, T and T' can reference each other on different columns. This can be accomplished by creating table T without a foreign key reference to T',

then creating table T' with a foreign key reference to T, and altering table T by adding a foreign key reference to T'. The “alter table add constraint” command is needed here because we cannot reference a table that does not yet exist, so we create table T without a foreign key reference, then add that constraint later, after creating table T'. In general, a referential cycle involves n tables, T_1 through T_n, such that T_1 includes a foreign key matching some candidate key in T_2, T_2 includes a foreign key matching some candidate key in T_3, and so on, ..., and T_n includes a foreign key matching some candidate key in T_1 [14]. The above algorithm can be modified to temporarily ignore one of the foreign keys involved in the cycle, thus breaking the cycle, and order the tables using the method described above. Thus, the State Generator can fill the tables except for the column whose foreign key constraint was ignored, and then “restore” the “missing” foreign key constraint (via the “alter table add constraint” command, as described above) so that we can fill those columns not yet filled. In other words, by performing another pass, we can handle a referential cycle.

If there is no cycle, then a topological sort algorithm, as described above, is sufficient to ensure that the State Generator populates referenced tables before tables that reference them. For the example schema in Figure 1.1a, the output of the topological sort based algorithm indicates that table `dept` must be filled before table `emp`.

3.3 Not null constraints

Handling not-NULL constraints is done as follows. Each attribute that does not have a not-NULL constraint is considered a candidate for NULL by the State Generator. Thus, a NULL group is implicitly added for this attribute in the Agenda DB. This informs the State Generator that it can choose NULL when generating a value for an attribute which is a candidate for NULL. Similarly, the Input Generator knows that it can instantiate an input parameter with a NULL value if the attribute associated with the parameter is a candidate for NULL. Further details on when the NULL value is selected are provided in Chapter 4.2.

3.4 Composite constraints

A constraint on multiple attributes is referred to as a composite constraint, or composite key constraint, as defined in Section 1.2. Attributes involved in composite constraints are marked as such in the Agenda DB so that they can be handled correctly during test generation.

3.4.1 Agenda Parser

The Agenda Parser stores in the Agenda DB's `composite_recs` table the following information for each attribute involved in a composite key constraint:

1. composite_index: unique index for this key
2. composite_type: CP (composite primary) or CU (composite unique) or CF (composite foreign)
3. table_name: the table which has this composite key
4. num_attributes: the number of attributes involved in this composite key
5. attr: attribute involved in this composite key

Consider table `district`, in the TPC-C schema (Appendix B), which has a composite primary key on attributes `d_id` and `d_w_id` indicating that no two warehouses in the same district have the same ID. The Agenda Parser inserts the following records in the Agenda DB table `composite_recs`:

- insert into composite_recs values(201, 'CP', 'district', 2, 'd_id');
- insert into composite_recs values(201, 'CP', 'district', 2, 'd_w_id');

In order to differentiate one composite key from another, AGENDA chooses a unique index (201, in the above example) for each composite key. By storing a tuple in `composite_recs` for each attribute involved in the composite key, AGENDA handles composite keys with an arbitrary number of attributes involved in the composite key.

3.4.2 State Generator

For attributes involved in composite primary/unique keys, the State Generator needs to select a unique combination of values.

We could store in the Agenda DB an entry for each combination of values for attributes comprising a composite primary/unique key; when a combination of values is selected, mark the combination as used, and check before selecting values for composite primary/unique key atributes to avoid selecting the same combination more than once. However, this method would be very inefficient, in terms of space, and could result in a combinatorial explosion. If there are n attributes involved in a composite primary/unique key and each attribute, *attr*, has V_{attr} values, then the number of combinations is $\prod_{attr=1}^{n} V_{attr}$. Combinatorial issues are discussed further in Section 4.1.

Instead, we use the procedure below, which does not explicitly store bookkeeping entries for the combinations that are used. The following procedure is repeated for each tuple inserted into the database, as long as unique combinations still exist for the values of the attributes in the composite primary/unique key.

- Call a sequencer method to retrieve the next unique combination. The sequencer gets the next combination by manipulating indices corresponding to the data groups of the involved attributes. This combination is used to select the data groups. There are n attributes involved in a composite key with m_i data groups for the i^{th} attributes. The sequencer generates n-tuples (i_1, ..., i_n) with 1 $\leq i_j \leq m_j$. Instead of storing all combinations generated, the sequencer only saves the number of groups for each attribute and the most recent combination generated in order to systematically generate the next combination. All combinations would be generated before the same combination is generated twice. For example, if there is a composite key on 3 attributes with 4, 3 and 2 data groups respectively, the first combination would be (1,1,1) meaning that the State Generator will select the first group for each attribute. Successive calls to the sequencer method would yield the following combinations, in sequence: (1,1,2), (1,2,1), (1,2,2), (1,3,1), (1,3,2), (2,1,1) (2,1,2), (2,2,1), (2,2,2), (2,3,1), (2,3,2), (3,1,1), (3,1,2), (3,2,1), (3,2,2), (3,3,1), (3,3,2), (4,1,1), (4,1,2), (4,2,1), (4,2,2), (4,3,1), and (4,3,2). The combination (3,1,2) means that the State Generator will select group 3 of the first attribute, group 1 of the second attribute, and group 2 of the third attribute.

- Using a seed which is unique for this composite key, pseudo-randomly choose a value in each data group (selected in the previous step) of the involved attributes.

While the above procedure is a reasonably efficient alternative to storing bookkeeping information

for every combination, there is a disadvantage; if the number of values for composite key attributes is limited and many combinations are generated, it is possible to generate a combination of values that was already used. When we forced AGENDA to generate 200 combinations, 5 were rejected. However, this cannot happen if the combination of groups is unique (as selected by the sequencer method above). Furthermore, our approach is based on the premise that a relatively small amount of data can encompass a rich variety of interesting situations. Reducing the amount of generated data is a primary goal of our approach, and software testing in general, as discussed in Section 4.1.

We now consider composite foreign keys. Suppose there is a composite primary key in table T_1 which is referenced by attributes in table T_2. For those attributes in T_2 involved in a composite foreign key, the State Generator chooses a combination of values already used in T_1. We identify three ways that this can be done:

1. Maintain bookkeeping information for each combination of values for each composite primary/unique key and use it to select the next unique tuple (if one exists) from the Agenda DB corresponding to the referenced table.

2. Using a cursor, select the next unique tuple (if one exists) from the referenced table T_1, and retrieve the values corresponding to the referenced attributes. Choose these values for the referencing attributes in the composite foreign key in T_2.

3. Utilize the same seed and sequencer method as was used in the generation of values for T_1. This approach does not work when attributes are involved in more than one composite key for the same table; for example, a table may have composite foreign keys CF(a,b,c) and CF(b,c,d); attributes b and c are involved in both composite constraints so if we use this method to select groups and values for (a,b,c) of the first composite foreign key, then we may have chosen values for b and c that do not work for the second composite foreign key. A similar problem occurs if an attribute is involved in both a composite primary key and a composite foreign key.

The first procedure is impractical, due to the issue of combinatorial explosion, as discussed above. To ensure correctness in all cases, the State Generator uses the second procedure, though conceivably the third could be used for attributes which are not involved in more than one composite key constraint.

3.4.3 Input Generator

The general procedure for the Input Generator is as follows:

1. Get heuristics for input generation.
2. While not done generating test cases
   (a) For each input parameter that needs to be instantiated
      i. Get relevant information (table, attribute) associated with this parameter from the Agenda DB.
      ii. Choose a data group and value according to the heuristics chosen.
      iii. Update bookkeeping information regarding progress achieved toward satisfying each heuristic.
      iv. Set done = true if all heuristics are satisfied, or if we cannot proceed (generated the maximum number of test cases).
   (b) Map the test case generated by step (a) to a template, in the Agenda DB.

If the Input Generator chooses values arbitrarily, then the data generated may cause at least one of the application's queries to return an empty table as the result. For example, suppose an application query has the following in its WHERE clause: emp.empno = :in_empno. If the Input Generator knowingly chooses a value for :in_empno for which the query will return an empty table, this is fine for testing the application's robustness (i.e. making sure that the application does not crash on a subsequent statement which accesses the empty table), but we also want to test the application on the kinds of data that would normally be expected. In this example, this means choosing a value for :in_empno which appears in the database for attribute `empno` of table `emp`. This would increase the likelihood that the transaction (defined in Section 1.2) under test commits/completes, thus exploring the application's handling of this scenario. A test case which is constructed with an effort to increase the likelihood of the transaction committing is classified as *type A*. Ideally, a *type A* test case satisfies all the WHERE clauses involving input parameters (if possible) so that if the transaction under test is correct, it will commit; if it does not commit, then there is an error in the transaction; if it commits, it may or may not be correct. A test case which is constructed with an effort to decrease the likelihood of the transaction committing is classified as *type B*. This thesis focuses on *type A* and discusses *type B* further below and in Chapter 6.

Generating *type A* test cases becomes harder when we consider composite constraints. For example, suppose an application query has the following in its WHERE clause: a = :input1 and b = :input2, and that attributes a and b are involved in a composite primary/unique/foreign key. For a *type A* test case, the Input Generator chooses values for input1 and input2 such that these values appear in the same tuple of the table on which the composite key on their associated attributes exists.

Independent parameters are input parameters which are directly associated with attributes involved in uniqueness or key constraints. Generating *type A* test cases is further challenged by *dependent parameters*, whose values depend on those of independent parameters. For example, suppose an application query has the following in its WHERE clause: c = :input3 and d = :input4, and that c has a primary key but d has no key constraint; the value chosen for input4 still depends on the value chosen for input3, in a *type A* test case. Furthermore, a parameter's value may depend on more than one attribute, if those attributes are involved in a composite key. The application discussed in Chapter 5 has many instances of input parameters associated with attributes involved in composite primary and/or composite foreign keys, as well as input parameters whose values are dependent on values selected for other input parameters (associated with attributes involved in primary key or composite constraints). Chapter 5 shows the results of AGENDA's operation on the TPC-C benchmark application.

The Input Generator handles composite keys and dependent parameters in generating *type A* test cases as follows:

1. Initialize
2. While not done generating test cases
    (a) Order the input parameters in the transaction under test so that parameters involved in key constraints (independent parameters) precede dependent parameters (as described above).
    (b) For each input parameter that needs to be instantiated
        i. Get relevant information (table, attribute) associated with this parameter from the Agenda DB.
        ii. Maintain bookkeeping information in a data structure (CompositeResults) so that the composite information (CompositeInfo) for the table associated with this parameter is efficiently stored and retrieved. Each unique CompositeInfo is pointed to by all the

parameters associated with the composite key attributes stored in this CompositeInfo. Each CompositeInfo also contains a pointer (cursor) to the next set of values to be used, which appear in the same tuple of the appropriate table (current table for a primary key, or referenced table for a foreign key).

iii. If there is a CompositeInfo for this parameter, then select a value v_0 such that when combined with values $v_1 ... v_{k-1}$ already selected for other parameters involved in the same composite key (on k attributes), values $v_0 ... v_{k-1}$ appear in the same tuple of the appropriate table.

iv. Else if the parameter's association to an attribute is direct (as defined in Section 2.6.1) and if the table associated with this parameter has a key on attribute(s) associated with other (independent) parameter(s), then this parameter is dependent on the values already selected; generate a query for deriving a value from the same tuple containing the appropriate values already selected.

v. Else select a value based on the other heuristics and constraints

(c) Map the test case generated by step (b) to a template, in the Agenda DB.

Figure 3.1 shows an example in which 2 input parameters to be instantiated are associated with attributes involved in a composite primary key, and ci_0 is a CompositeInfo object representing this composite primary key, as discussed above. This object's nextTuple field, via a database cursor, iterates over the tuples of table t, which has a composite primary key constraint on attributes a and b. In Figure 3.1, the next tuple is the second row of table t, with values of 4 and 7 for attributes a and b respectively. Since a and b are associated with input parameters in_a and in_b, generation of a *type A* test case means that selection of value 4 for in_a implies that 7 must be selected for in_b, since these values appear in the same tuple of table t and thus this combination of values would be considered a normal input by the application under test. The field numCPDataGenerated of ci_0 indicates the number of data generated in the current test case for this composite key; the field numParamsAssocWithCP of ci_0 indicates the number of input parameters associated with attributes involved in this composite key; when these 2 fields are equal, then numCPDataGenerated is reset to 0 and nextTuple iterates so that it points to the next tuple.

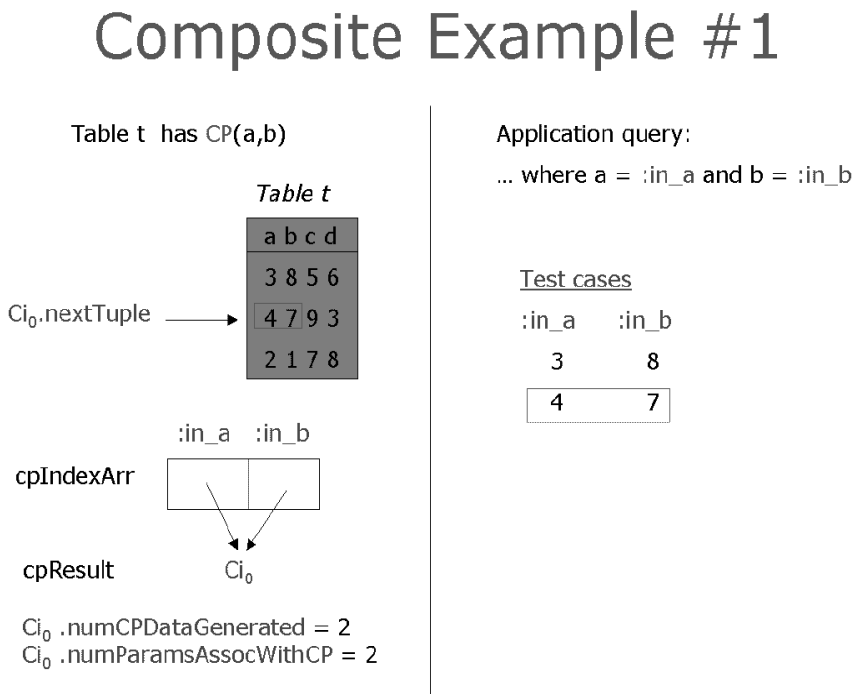


Figure 3.1: Composite Example 1

Figure 3.2 shows an example with the same composite information as the previous example. Input parameters in_a and in_b are associated with attributes a and b respectively of a composite primary key constraint on table t. However, there is an additional input parameter in_c associated with attribute c of table t. The value chosen for in_c depends on the values chosen for in_a and in_b, when generating a *type A* test case. A query is generated to derive the value of dependent parameter in_c based on the values already chosen for in_a and in_b for the current test case. In Figure 3.2, the value chosen for in_c will be 9, since that is the value of c in the same tuple as values 4 and 7, which were already selected for in_a and in_b.

Figure 3.3 shows an example with some input parameters associated with attributes involved in a composite constraint on one table, and other input parameters associated with attributes involved in a composite constraint on another table. Input parameters in_a, in_b and in_c are associated with attributes a, b and c, respectively, of a composite primary key constraint on table *t1*. Input parameters in_x and in_y are associated with attributes x and y, respectively, of a composite primary key constraint on table *t2*. In addition, input parameter in_d is dependent on the values chosen for in_a, in_b and in_c, and in_z is

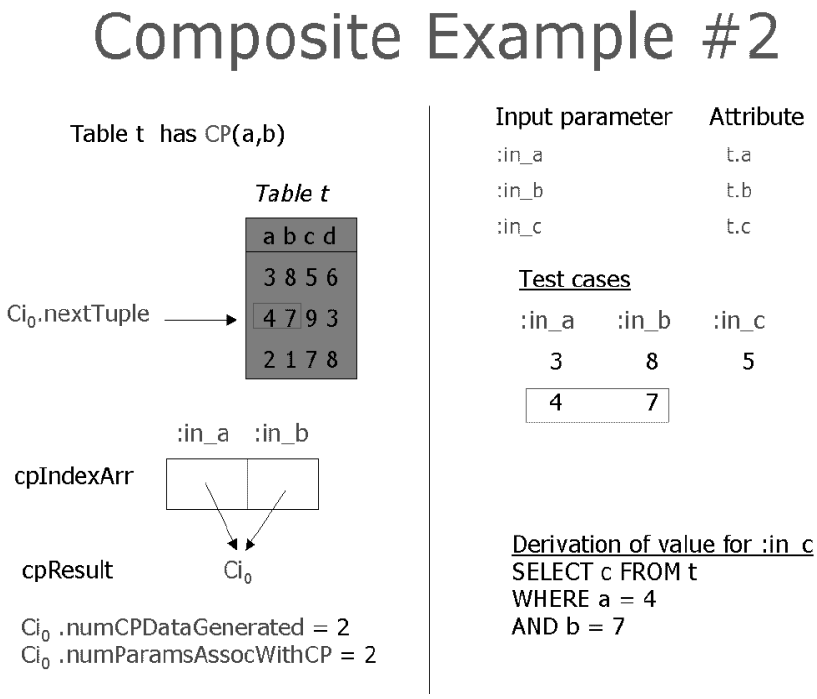


Figure 3.2: Composite Example 2

dependent on the values chosen for in_x and in_y. This example illustrates the ordering done in step (2a) above. Dependent parameters in_d and in_z are processed after those parameters on whose values they depend. This example also illustrates that parameters associated with key attributes can be processed in an arbitrary order, since a mapping is maintained between each of the input parameters to be instantiated and the appropriate CompositeInfo object (if any) associated with this parameter. In this example, there are 2 CompositeInfo objects, ci_0 and ci_1, since there are 2 composite keys.

Figure 3.4 shows an example with input parameters associated with attributes involved in both a composite primary key and a composite foreign key. Input parameters in_x and in_y are associated with attributes x and y, respectively, of a composite primary key and a composite foreign key on table *t2*. By leveraging the work already done by the State Generator in filling table *t2* with data that satisfy the composite foreign key constraint, the Input Generator need not consider the composite foreign key when choosing values for in_x and in_y. Data values chosen for in_x and in_y from the CompositeInfo object for the composite primary key on attributes x, y and z implicitly satisfy the composite foreign key constraint on attributes x and y. It might appear that composite foreign key information can always be

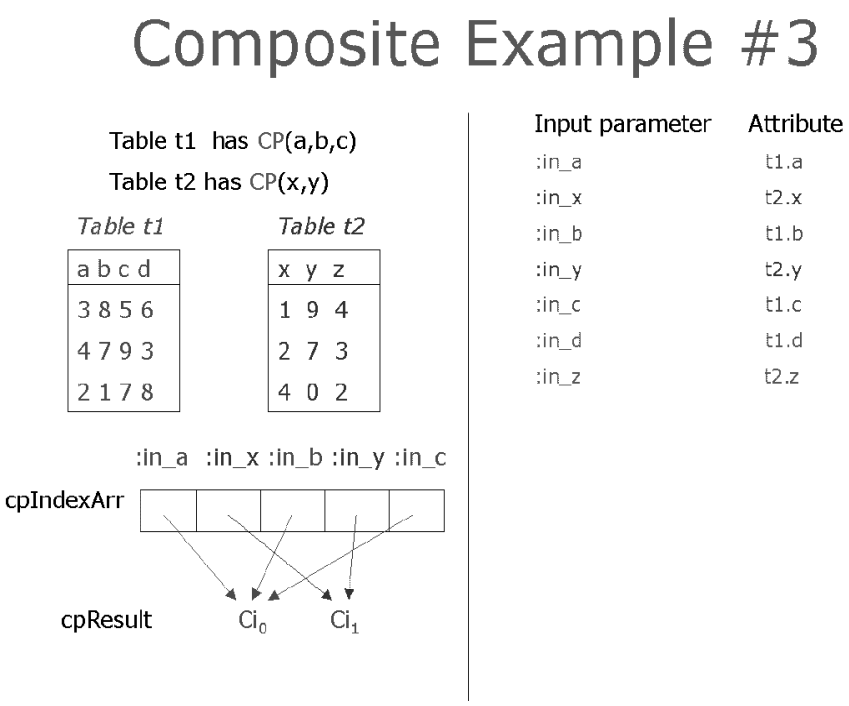


Figure 3.3: Composite Example 3

ignored; however, suppose that the application queries do not access tables involved in composite primary key constraints, but only access tables involved in composite foreign key constraints; in this case, the input parameters will be associated with composite foreign key attributes; handling of composite foreign keys is similar to the handling of composite primary keys, except that data values are chosen from the referenced table, and bookkeeping for multiple composite foreign keys must be done for tables with multiple composite foreign keys.

Classifying test cases also gives the validation tools (and ultimately the user) a better understanding of which test cases are more or less likely to cause the application to fail. The goal is to provide the user with more useful feedback. For example, if the application fails to complete only on *type B* test cases, this indicates a problem with the application's robustness. If the application fails to complete on a *type A* test case, then there is a problem with the application's correctness.

Composite Example #4

- Attributes involved in CP and CF

Table t1

a	b	c	d
3	8	5	6
4	7	9	3
2	1	7	8

Table t2

x	y	z
3	8	4
4	7	2
2	1	6

- T1: CP(a, b)
- T2: CF(x, y) references T1, CP(x, y, z)

Input parameter	Attribute
:in_x	t2.x
:in_y	t2.y
:in_z	t2.z

Test cases

:in_x	:in_y	:in_z
3	8	4
4	7	2
2	1	6

Figure 3.4: Composite Example 4

Chapter 4

Data Generation

4.1 Combinatorial Issues

4.1.1 Partitioning of data

Values and groups

The approach is loosely based on the Category-Partition method [33]: the user supplies suggested values for attributes, partitioned into groups, which we call *data groups*. This data is provided in the sample-values files. AGENDA produces meaningful combinations of these values in order to fill database tables and provide input parameters to the application program, while assuring that the constraints are satisfied. Data groups are used to distinguish values that are expected to result in different application behavior, e.g. different categories of employees.

For example, the tester might provide the file shown in Figure 4.1a (among others) for testing the application whose schema and queries are shown in Figure 1.1. In the file `empno`, the tester has supplied fifteen possible values, partitioned into three data groups, student, faculty, and administrator. More details on how these values are selected for insertion are provided in Chapter 3 and Section 4.2.

We have deliberately chosen this partially-automated generation approach, rather than attempting to automatically generate attribute values satisfying the integrity constraints. Checking test results cannot be fully automated (unless a complete formal specification is available) and we believe it will be easier for the human checking the test results to work with meaningful values, rather than with randomly generated values. We also explored a hybrid approach, described below, in which automatic generators

a)

empno:
–choice_name: student
111
112
113
114
115
—-
–choice_name: faculty
550
555
565
569
570
—-
–choice_name: administrator
811
812
813
814
815

b)

```
insert into dept values(10,'NULL','Brooklyn');
insert into dept values(20,'accounting','NULL');
insert into dept values(30,'research','Athens');
insert into dept values(40,'sales','Florham Park');
insert into emp values(111,'Smith',NULL,50.00,10);
insert into emp values(550,'Jones',6000.00,NULL,20);
insert into emp values(811,'Blake',6000.01,500.00,30);
insert into emp values(112,'Clark',6056.29,1000.00,40);
insert into emp values(555,'Adams',6999.99,2000.00,10);
insert into emp values(812,'Davis',7000.00,10000.00,20);
insert into emp values(113,'Flanders',7000.01,50.00,30);
insert into emp values(565,'Martinez',7027.52,500.00,40);
insert into emp values(813,'Williams',8999.99,1000.00,10);
insert into emp values(114,'Fox',9000.00,2000.00,20);
insert into emp values(569,'Rivera',9000.01,10000.00,30);
insert into emp values(814,'Hernandez',9175.45,50.00,40);
insert into emp values(115,'Ullman',9255.68,500.00,10);
insert into emp values(570,'White',9999.99,1000.00,20);
insert into emp values(815,'Widger',10000.00,2000.00,30);
```

Figure 4.1: Sample: a) input file for `empno` attribute and b) insertions produced by the tool

are associated with some domains (e.g. integers within a given subrange) and testers supply possible values for other domains.

Automatic Partitioning Into Data Groups

Usability is one of the main design goals of our tool set. In order to populate a database state with meaningful values, we require the tester to provide a sample of such values for each attribute, in the form of an input file optionally partitioned into data groups. While this may be helpful and appropriate for attributes such as `ename` in table `emp`, it could be burdensome for other attributes, particularly numeric ones of type integer, float, or money. In order to improve the usability of our tool, we have designed and implemented a mechanism for automatic derivation of input files, divided into meaningful data groups, for certain input types. Suppose, as in Figure 1.1, for some attribute, `salary`, there is a check constraint in the schema: "check(`salary` $\geq$ 6000.00 and `salary` $\leq$ 10000.00)" and there are two application queries, one whose clause contains "`salary` $\geq$ 5000.00 and `salary` $\leq$ 10000.00" and the other whose clause contains "`salary` $>$ 7000.00 and `salary` $\leq$ 9000.00". This suggests that the intervals $[5000.00, \ldots, 6000.00)$, $[6000.00, \ldots, 7000.00]$, $(7000.00, \ldots, 9000.00]$, and $(9000.00, \ldots, 10000.00]$ might be treated differently and should constitute different data groups.

In addition, points on and near boundaries are believed to be particularly likely to be handled incorrectly, and therefore likely to cause failures, so it is useful to force the inclusion of such values. These

are known as ON and OFF points in the domain testing strategy introduced by White and Cohen [41]. For clarity, we further distinguish between Interior-OFF points and Exterior-OFF points depending on whether or not the point satisfies the constraint. For now, we assume the tester will wish to select one ON point and two OFF points, one interior and one exterior. Weyuker and Jeng developed a variant of domain testing in which only one ON point and a single very close exterior OFF point are selected for each boundary [40].

The tester could manually generate values for the `salary` attribute, and partition them into data groups. This would probably involve examining the schema and DB application while trying to identify boundary values for this attribute as well as near-boundary values. In this example, the boundary values are 5000.00, 6000.00, 7000.00, 9000.00, and 10000.00. The tester would then determine which ON and OFF points to use. For example, 7000.00 is an ON point because it defines the border. This is true even though it does not satisfy the constraint "check(`salary` $>$ 7000.00)". 7000.01 is an OFF point because it does not lie on the border.

Finally, the tester would create an input file such as the one for `salary` in Figure 4.2. However, creating the input file in this manner is very tedious for the tester. Since this process is automatable, we have added this functionality to our tool. If values associated with a specific attribute can be extracted from the schema and/or the DB application's queries by the Agenda Parser, then the tester is given the option of letting the tool automatically partition these values into data groups. Otherwise, the tester is asked to supply possible values. Partitioning is done by treating the values extracted from the schema and application as boundary values. Groups containing each of these values by themselves as well as separate groups containing intermediate and off-by-one values (denoted as ON and OFF points) are automatically generated as shown in Figure 4.2.

The intermediate values are randomly generated in the intervals between the border values. Currently, the tool can automatically partition attributes of integer, float, and money types, and can handle simple expressions like those in the example. We plan to explore in the future what can be done for attributes of type string, and more complicated semantic expressions in the schema and/or application, such as "`salary` $\leq$ `dept-head-salary * 0.50`".

Another future consideration is the handling of data outside the range allowed by the schema. We can give the tester the option of deciding whether to allow data that should fail. If the DBMS is not enforcing constraints, it might be useful to include such data. In our example, data of this nature are salaries

between 5000.00 and 5999.99 because the schema permits salaries between 6000.00 and 10000.00 yet one of the application's queries asks for salaries greater than or equal to 5000.00.

```
--choice_name: exterior_OFF_point_1
4999.99
--choice_name: ON_boundary_value_1
5000.00
--choice_name: interior_OFF_point_1
5000.01
--choice_name: interboundary_values_1
5168.40
5310.53
--choice_name: exterior_OFF_point_2
5999.99
--choice_name: ON_boundary_value_2
6000.00
--choice_name: interior_OFF_point_2
6000.01
--choice_name: interboundary_values_2
6056.29
6230.12
--choice_name: exterior_OFF_point_3
6999.99
--choice_name: ON_boundary_value_3
7000.00
--choice_name: interior_OFF_point_3
7000.01
--choice_name: interboundary_values_3
7027.52
7322.28
--choice_name: interior_OFF_point_4
8999.99
--choice_name: ON_boundary_value_4
9000.00
--choice_name: exterior_OFF_point_4
9000.01
--choice_name: interboundary_values_4
9175.45
9255.68
--choice_name: interior_OFF_point_5
9999.99
--choice_name: ON_boundary_value_5
10000.00
--choice_name: exterior_OFF_point_5
10000.01
```

Figure 4.2: File generated for `salary` attribute

4.1.2 Reducing the amount of data generated

Using these data groups and guided by heuristics selected by the tester, AGENDA produces a collection of *test templates* representing abstract test cases. The tester then provides information about the expected behavior of the application on tests represented by each template. For example, the tester might specify that the application should increase the salaries of employees in the "faculty" group by 10% and should not change any other attributes.

In order to control the explosion in the number of test templates and to force the generation of particular kinds of templates, the tester selects heuristics. Available heuristics include one to favor "boundary values", heuristics to force the inclusion of NULL values where doing so is not precluded by not-NULL constraints, heuristics to force the inclusion of duplicate values where so doing is not

precluded by uniqueness constraints, and heuristics to force the inclusion of values from all data groups. These are described in more detail in the next chapter.

Finally, AGENDA instantiates the templates with specific test values, executes the test cases and checks that the outputs and new database state are consistent with the expected behavior indicated by the tester.

4.2 Heuristics

4.2.1 Purpose

Heuristics, described below, are selected by the tester to guide the generation of both the DB state and the input test cases for the application. The purpose is twofold:

1. Reduce the amount of test data generated.

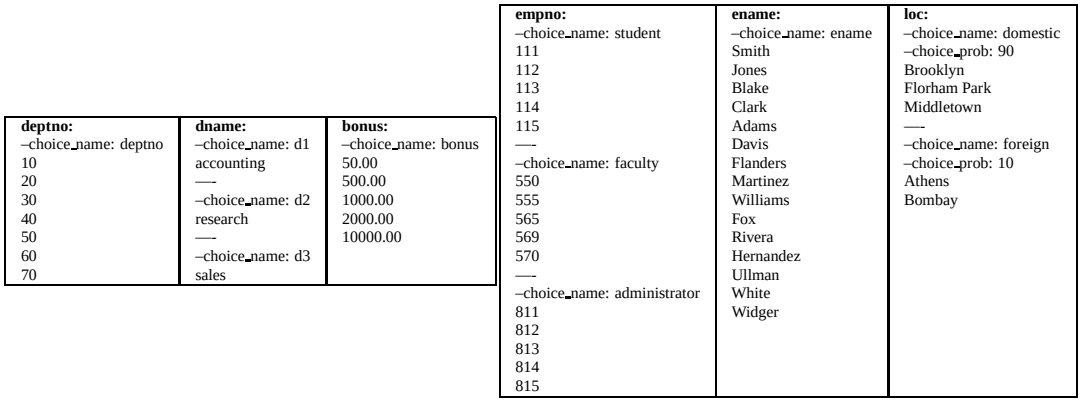
2. Expose likely faults in the application.

4.2.2 Heuristics for State Generation

The State Generator prompts the user to select the desired heuristic(s) for state generation. The available heuristics to guide state generation are: `boundary values`, `duplicates`, `nulls`, and `all groups`.

As discussed earlier, constant values that are associated with attributes in the application's DB schema and queries, such as the values 5000.00, 6000.00, and 10000.00 for the attribute `salary` in Figure 1.1, can be very useful in that they define boundaries. These `boundary values` are extracted and stored in the Agenda DB by the Agenda Parser. An attribute is populated with boundary values if the corresponding heuristic is chosen and the State Generator finds boundary values, in the Agenda DB, associated with the attribute.

If the user selects the `duplicates` heuristic, then the State Generator fills the application DB with duplicate values for those attributes that have no uniqueness constraints. The user may enter the number of duplicates or may indicate that the State Generator should insert a duplicate value for every non-unique attribute.



deptno:	dname:	bonus:
–choice_name: deptno	–choice_name: d1	–choice_name: bonus
10	accounting	50.00
20	—-	500.00
30	–choice_name: d2	1000.00
40	research	2000.00
50	—-	10000.00
60	–choice_name: d3	
70	sales	

empno:	ename:	loc:
–choice_name: student	–choice_name: ename	–choice_name: domestic
111	Smith	–choice_prob: 90
112	Jones	Brooklyn
113	Blake	Florham Park
114	Clark	Middletown
115	Adams	—-
—-	Davis	–choice_name: foreign
–choice_name: faculty	Flanders	–choice_prob: 10
550	Martinez	Athens
555	Williams	Bombay
565	Fox	
569	Rivera	
570	Hernandez	
—-	Ullman	
–choice_name: administrator	White	
811	Widger	
812		
813		
814		
815		

Figure 4.3: Input files for Department-Employee database

If the tester selects the `nulls` heuristic, then the State Generator selects nulls for those attributes that can be null. If an attribute has no uniqueness constraint and no not-NULL constraint, then the attribute is a candidate for a null value. If the tester does not specify the number of nulls to include in the DB state, a null value will be selected once for each candidate.

If the tester selects the `all groups` heuristic, then all data groups, associated with all attributes of all the application's tables, are represented among the tuples generated for the application DB state.

Information in the Agenda DB is shown in sub-section 2.6.2.

4.2.3 Example for State Generation

Here, we present an example illustrating a database state produced by the tool for the Department-Employee schema shown in Figure 1.1a. Input files supplied by the tester are shown in Figure 4.3. The file for the `salary` attribute, automatically generated as described in sub-section 4.1.1, is shown in Figure 4.2.

The tester is prompted to select the desired heuristic(s) to guide state generation. Depending on the tester's choices, the generated state may include, where appropriate, nulls, duplicates, boundary values, and representation of all data groups. The tester may select more than one heuristic.

Suppose that the tester has chosen `nulls`, `boundary values` and `all groups`.

If the tester does not specify the number of tuples to generate, state generation will continue until all heuristics have been satisfied or all values for a unique attribute have been exhausted.

The populated tables are shown in Figure 4.4. These have been obtained by executing the `insert` statements generated by the tool (see Figure 4.1b) and then outputting the resulting database state.

table dept

deptno	dname	loc
10	NULL	Brooklyn
20	accounting	NULL
30	research	Athens
40	sales	Florham Park

table emp

empno	ename	salary	bonus	deptno
111	Smith	NULL	50.00	10
550	Jones	6000.00	NULL	20
811	Blake	6000.01	500.00	30
112	Clark	6056.29	1000.00	40
555	Adams	6999.99	2000.00	10
812	Davis	7000.00	10000.00	20
113	Flanders	7000.01	50.00	30
565	Martinez	7027.52	500.00	40
813	Williams	8999.99	1000.00	10
114	Fox	9000.00	2000.00	20
569	Rivera	9000.01	10000.00	30
814	Hernandez	9175.45	50.00	40
115	Ullman	9255.68	500.00	10
570	White	9999.99	1000.00	20
815	Widger	10000.00	2000.00	30

Figure 4.4: A database state produced by the tool

Small table sizes were chosen to illustrate the concepts; AGENDA can generate much larger tables. The database state produced by the State Generator satisfies all of the constraints in the schema and all of the heuristics selected by the tester. Also, the data groups are represented in accordance with the probability annotations given in the input files. For example, in table `emp`, `deptno` has a foreign key referencing table `dept`. Values selected for `emp.deptno` are chosen among those used in `dept.deptno`. Only one foreign city was selected for the `loc` attribute because the choice probability for foreign cities is only 10%, as specified in the input file called `loc`. Data groups for other attributes are represented more equally, since the default choice probability is 100% divided by the number of choices (data groups). [1]

4.3 Heuristics for Input Generation

The Input Generator generates test cases by instantiating the application's inputs with actual values. For now, we only consider test cases for a single parameterized application query. A test case for such a

[1]Note that, although this is not the main purpose of the tool, it also exposes a possible flaw in the schema: since no not-NULL constraints exist for `dname`, `loc`, `salary` and `bonus`, NULL values were selected for those attributes; examining the tables may lead the tester to question whether those attributes should have had not-NULL constraints, or execution of test cases accessing those rows may cause unexpected behavior.

query consists of instantiating all input parameters of the query. The input variables are clearly marked as such in each application query to be tested. For example, a colon precedes an input parameter in embedded SQL, as in the queries in Figure 1.1b.

In order to generate a test case, each input parameter must be instantiated. The Input Generator prompts the tester to select the desired heuristic(s) to guide test case generation. The available heuristics to guide input generation are: `boundary values`, `duplicates from the application DB state`, `nulls`, `all groups`, and `all templates`. The tester may select more than one heuristic. For each test case, the Input Generator chooses a value for each input parameter, among those supplied by the tester and stored in the Agenda DB by the Agenda Parser, with guidance from the selected heuristics.

As discussed earlier, constant values that are associated with attributes in the application's DB schema and queries, such as the values 5000.00, 6000.00, and 10000.00 for the attribute `salary` in Figure 1.1, can be very useful in that they define boundaries. These `boundary values` are extracted and stored in the Agenda DB by the Agenda Parser. An input parameter is instantiated with boundary values if the corresponding heuristic is chosen and the Input Generator finds boundary values, in the Agenda DB, associated with the input parameter. Boundary values may also be defined in the sample-values files, by assigning each boundary value to its own data group; selection of the `all groups` heuristic, defined below, would ensure that each data group, and thus each boundary value, is represented among the data generated. This applies to the State Generator as well.

If the state generation was guided by the `duplicates` heuristic, then the generated application database will contain some tuples that on a specific attribute will have the same value. By selecting the `duplicates` heuristic for the Input Generator, the tester indicates that the test cases should include some of those values that appear more than once on a specific attribute in the application DB. For example, suppose three different tuples were generated by the State Generator with the values 111, 111, and 222 respectively on a specific attribute. The heuristic `duplicates` for the Input Generator means that a value like 111, which appears more than once in the application DB, should be chosen. This value need not appear in more than one of the test cases.

If the tester selects the `nulls` heuristic, then the Input Generator selects nulls for those parameters that can be null. If the attribute associated with the parameter has no uniqueness constraint and no not-NULL constraint, then the parameter is a candidate for a null value. The number of nulls included among the generated test cases is decided by the tester; the tester may enter a number or may indicate

that the Input Generator should instantiate each parameter that can be null with a null value. By default, the maximum number of nulls appearing in a single test case is one.

If the tester selects the `all groups` heuristic, then all data groups, associated with the input parameters, are represented among the test cases generated. Each unique combination of data groups comprises a test template. If the tester selects the `all templates` heuristic, then all test templates are represented among the test cases generated. After generating each test case, the Input Generator maps the test case to a specific template that defines the expected output. This information is stored in the Agenda DB and used by the validation tools. Test templates are discussed in more detail in the example below.

The implementation of the Input Generator is similar to that of the State Generator in that it uses the constraints from the schema and the heuristics selected by the tester to guide the generation. The Input Generator is ready to run after the Agenda Parser has filled the Agenda DB, the State Generator has populated a DB state as in Figure 4.4, and the tester has selected the desired heuristics. The tester is prompted for the number of test cases to generate. If the tester does not specify the number of test cases, then test cases are generated until all selected heuristics have been satisfied or all possible values for a unique attribute have been exhausted.

Some test cases may satisfy more than one heuristic at the same time. In order to perform the test case generation in accordance with the tester's selections more efficiently, we mark in the Agenda DB, the groups and values that are used, while generating test cases based on each heuristic chosen, and we keep track of the progress made in satisfying each heuristic, so that the tool need not generate test cases for a heuristic that was already satisfied. Since the tester may also select more than one heuristic to guide state generation, this applies to the State Generator as well.

```
--choice_name: low
1.01
1.05
1.07
----
--choice_name: high
1.50
1.75
```

Figure 4.5: Input file for `rate` parameter of `update` query in Figure 1.1b

4.3.1 Example for Input Generation

The Input Generator generates test cases by instantiating values for the application query's parameters, or input host variables. The `update` query in Figure 1.1b has two input parameters: `rate` and `in_empno`. Suppose the tester selects `all groups` as the heuristic to guide the Input Generator. The Input Generator queries the Agenda DB for information about each input host variable. It finds that `in_empno` is directly associated with attribute `empno` of table `emp` and that the tester supplied sample values for `empno` partitioned into 3 groups: student, faculty, and administrator, as in Figure 4.3. It then finds that `rate` is not directly associated with any attribute but the tester has supplied sample values, partitioned into 2 groups, low and high, as in Figure 4.5. Since there are two input parameters, one with 3 groups and the other with 2 groups, there are a total of 6 test templates for test case generation: (student,low), (student,high), (faculty,low), (faculty,high), (administrator,low) and (administrator,high).

Since the tester selected `all groups` as opposed to `all templates`, the Input Generator knows that it is sufficient for each group to be represented among the test cases, rather than all combinations of all groups. This is accomplished by marking groups that are used in test cases in the Agenda DB, just as the State Generator marks groups that are used in the DB state, and giving preference to the least frequently used group when instantiating its associated parameter, thus reducing the number of test cases needed to satisfy all groups. For this example, only 3 test cases are needed to satisfy all groups: (student,low), (faculty,high), (administrator,low). Sample test cases produced by the Input Generator for this application query are: (111,1.01), (550,1.50), (811,1.05).

Chapter 5

Case Study based on TPC-C benchmark application

TPCBenchmark^{TM} C (TPC-C) is the standard benchmark for online transaction processing. It is a mixture of read-only and update-intensive transactions that simulate the activities found in complex OLTP [37]. Although the TPC-C benchmark application was designed for database performance testing, we chose this application for our case study, in order to exercise our tool set on a real application with a complex schema. The TPC-C schema has multiple composite primary and composite foreign key constraints, some tables have both a composite primary key and one or more composite foreign keys, and some attributes are involved in both a composite primary key and a composite foreign key. AGENDA handled all of these constraints in generating *type A* test cases, as discussed in Chapter 3.

The TPC-C application models a wholesale supplier with a number of geographically distributed districts and associated warehouses. There are 9 tables (warehouse, district, customer, history, new_order, c_orders, order_line, item, and stock) and 5 transactions (new-orders, payment, order-status, delivery, and stock-level). For the purpose of this case study, each transaction is considered as a separate program.

The TPC-C schema is provided in Appendix B. The TPC-C tables are shown in Figure 5.1 as a graph, in which each node represents a table and each directed edge (T, T') indicates that there is a foreign key constraint on one or more attributes of table T referencing table T'. The number in parenthesis next to each table name in Figure 5.1 indicates when this table can be filled, according to the output of the

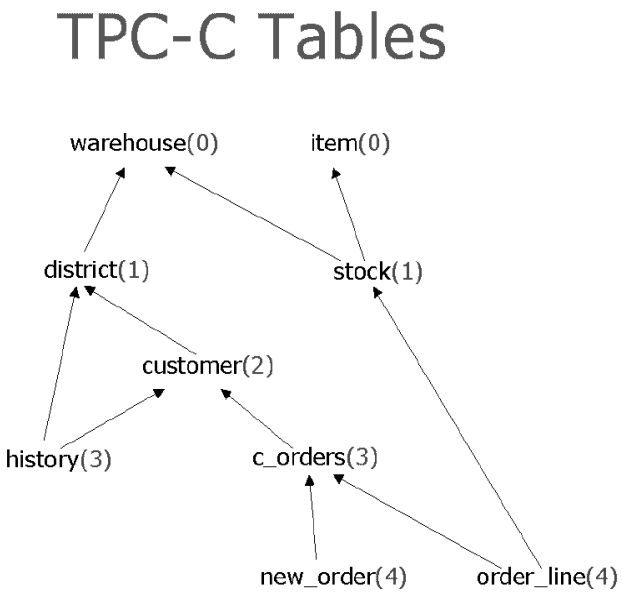


Figure 5.1: TPC-C tables

topological sort based algorithm discussed in Section 3.2.2, ensuring that a referenced table is filled before any table that references it.

A graduate student who is an experienced database application programmer created buggy versions of the TPC-C transactions by seeding a diverse set of SQL errors that he considered common and realistic. A description of each seeded error is provided in Figure 5.2. Each buggy implementation contained a single error, among E1 through E11, and test data was generated for each implementation separately, since in reality, we do not have the correct implementation.

For each transaction of the TPC-C application, AGENDA generated a database state (45 rows for 9 tables; 5 rows per table; all data groups represented) and generated test cases (according to the *type A* and `all groups` heuristics); then, each test case generated for the correct implementation was executed on that implementation and each test case generated for 11 buggy versions (with errors labelled E1, E2, E3, ..., E11 respectively, as in Figure 5.3) was executed on the transactions with each of those errors.

The reason for generating test cases for the correct implementation was to check that the tool was indeed producing *type A* test cases. There were no false positives, in that all of the test cases generated

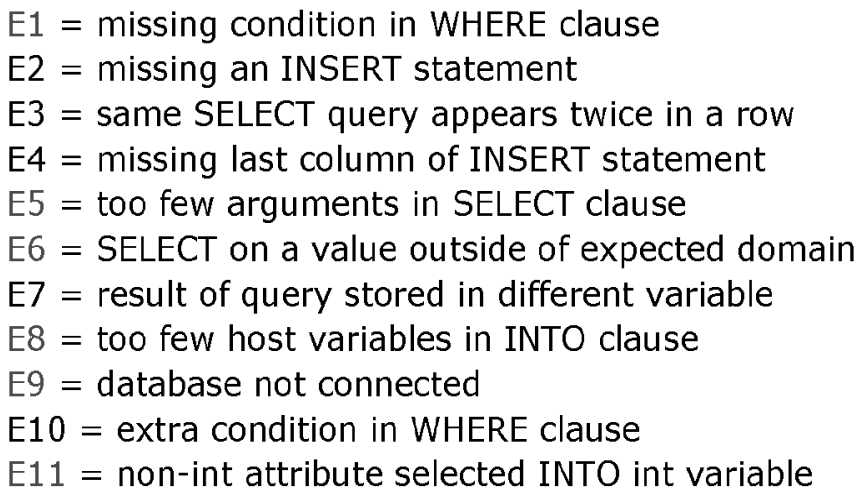


Figure 5.2: Seeded errors

for the correct implementation, when executed on the correct implementation, resulted in the transaction committing (in accordance with the definition and purpose of *type A*). When a *type A* test case is executed by an application under test (buggy implementation) and the transaction does not commit, then we know there is an error in the implementation. In Figure 5.3, an entry of ”Y” means that the test case exposed an error, since the application’s transaction did not commit on a *type A* test case. An entry of ”N” means that the buggy transaction did commit, and thus the error was not exposed. The last column provides summary information, indicating whether or not at least one test case exposed a particular error.

Error E9 is an anomalous error, since the application, executing with this error (database not connected), cannot commit regardless of the test case. All other errors were seeded by modifying SQL queries in the TPC-C application. Only one test case was produced for the transactions with errors E7, E10 and E11 because each input parameter of these transactions was associated with an attribute having just one data group; thus all groups could be represented by a single test case. When the tool was forced to produce additional *type A* test cases, the results did not improve; they stayed the same, as in Figure 5.4. Furthermore, heuristics other than `all groups` and *type A* did not improve the results. The

Error	Test Case 1	Test Case 2	Test Case 3	Test Case 4	at least 1 test case
E1	Y	Y	Y	-	Y
E2	N	N	N	-	N
E3	N	N	N	-	N
E4	N	N	N	-	N
E5	N	N	N	Y	Y
E6	N	N	N	Y	Y
E7	N	-	-	-	N
E8	N	N	N	Y	Y
E9	Y	Y	Y	Y	Y
E10	N	-	-	-	N
E11	Y	-	-	-	Y

Figure 5.3: Case study results

Error	Test Case 1	Test Case 2	Test Case 3	Test Case 4	at least 1 test case
E7	N	N	N	N	N
E10	N	N	N	N	N
E11	Y	Y	Y	Y	Y

Figure 5.4: More case study results

`nulls` heuristic is not appropriate for this experiment because NULL values are not considered normal inputs by the TPC-C application, and thus the `nulls` heuristic is appropriate for testing robustness (*type B*) rather than correctness (*type A*). Since the TPC-C application was not designed to be robust, it fails when an input parameter has a NULL value or any value that does not occur in the database.

At first glance, it might appear that for some SQL errors (such as syntax errors) that occur on all paths, any test case would expose the fault. However, if the transaction under test has n queries and there is an error in the k-th query, then a test case, in order to expose this error, must pass through the first k-1 queries. Moreover, if the error occurs on some (not all) path(s), as in E5, E6, and E8, then in order to expose the fault, some input test case needs to be generated that forces execution of the path containing the error. Errors E5, E6 and E8 were exposed by at least one test case due to appropriate partitioning of data groups, as well as the tool's ability to represent all groups and generate *type A* test cases. Error E1 involved a missing condition in a WHERE clause; the missing condition contained an attribute involved in a composite constraint, thus causing a subsequent composite constraint violation, given AGENDA's test cases.

The following errors were not exposed: E2, E3, E4, E7 and E10. The transaction with error E3 is not

incorrect, but inefficient; the same select query is executed twice in succession. Errors E2, E4, E7 and E10 involve semantic errors (missing an insert statement, missing the last column in an insert statement, storing the result of a select query in a different variable, and an extra condition in the WHERE clause, respectively). Though error E1 (missing condition, as described above) was exposed, in general, the parsing and generation tools (Agenda Parser, State Generator and Input Generator) cannot by themselves expose semantic errors; however, by incorporating preconditions and postconditions in conjunction with the test data generated, the validation tools (State Validator and Output Validator) can expose semantic errors when a property is violated. AGENDA's validation tools exposed errors E2, E4, and E7. Further details on the operations of the validation tools are provided in [17].

5.1 Costs of running the tools

The costs of executing AGENDA's tools on 5 examples are shown in Figure 5.5. Each entry is an average over 5 executions. The setup time consists primarily of removing old information (from prior runs) from the Agenda DB tables. The checking time includes the execution time for running the test cases and the time for validating the results. The total numbers of rows and test cases produced by the State Generator and Input Generator, respectively, are provided. All experiments were performed on the platform of DELL Dimension 811. The CPU is a Pentium 4 Processor at 1.3 GHz. Main memory is 256 MB. The system is implemented as a web database application. The overhead includes the additional cost of automatic execution of the AGENDA application as specified in XML configuration files. Summary information is provided in the last 3 rows of Figure 5.5: time per row to generate the Application DB, time per test case to generate the inputs, and time per test case to check the results.

Additional information about each of the examples is provided in Figure 5.6: number of tables, number of attributes, number of constraints (primary key, foreign key, unique, composite key, not NULL, and check), number of data groups per attribute, number of input parameters, number of data groups per parameter, and number of test templates. The number of test templates is the product of the number of data groups for each parameter. For example, for *emp2*, there are 2 parameters, whose data values are partitioned into 21 groups and 12 groups respectively. Therefore the number of test templates is 252, the product of 21 and 12.

	cia	movie	emp	emp1	emp2
Setup time (ms)	6820	6156	7520	7099	8190
Agenda Parser time (ms)	12767	16352	17132	16733	18692
State Gen. time (ms)	2665	9589	6324	8216	19534
Input Gen. time (ms)	2281	1948	2883	11064	177221
Checking time (ms)	3309	4736	5476	25316	145342
Number of rows generated	3	30	12	15	29
Number of test cases generated	2	1	3	42	252
State Gen. time / row (ms)	888	320	527	548	674
Input Gen. time / test case (ms)	1140	1948	961	263	703
Checking time / test case (ms)	1654	4736	1825	603	577

Figure 5.5: Time (in milliseconds) for running AGENDA's tools

Number of	cia	movie	emp	emp1	emp2
tables	1	3	2	2	2
attributes	4	11	8	8	8
primary key constraints	1	2	2	2	2
foreign key constraints	0	2	1	1	1
unique constraints	0	0	1	1	1
composite key constraints	0	1	0	0	0
not NULL constraints	0	0	1	1	1
check constraints	0	1	0	0	0
data groups per attribute	3,1,1,1	1	1,3,2,3,1,1,1,1	1,3,2,7,1,1,1,1	1,3,2,21,1,1,1,1
input parameters	1	2	2	2	2
data groups per parameter	2	1, 1	3, 2	7, 6	21, 12
test templates	2	1	6	42	252

Figure 5.6: Additional information about the examples

5.2 Limitations

The Agenda Parser accepts any database schema (full SQL), since the core of the tool is the DBMS SQL parser, but the current tool set implementation does not take full advantage of certain SQL constructs which may contain useful information for the purpose of testing. For the purpose of generating test data for the database state and test cases for the application, AGENDA utilizes all uniqueness constraints, referential integrity constraints, not-NULL constraints, composite constraints, and semantic constraints to a limited extent (for extracting boundary values, as discussed in Section 4.1.1) but not complicated semantic and domain constraints. The current tool set implementation extracts useful information from individual queries, such as the input and/or output parameters , the table and attribute

with which each parameter is directly/indirectly associated, the type of query and which table is potentially changed by the query if it is an UPDATE or INSERT. Since source code analysis and a data flow graph are not yet part of the tool set, it is assumed for now that the user supplies information about data flow and which input parameters need to be instantiated for which queries. Some input parameters do not need to be instantiated because their values are determined by statements executed before the query under test. Recently, AGENDA has been extended to handle more complicated semantic constraints, in the context of testing database transaction consistency and concurrency [17, 18]. Also, the AGENDA prototype system has been demonstrated [6].

Chapter 6

Conclusions and Future Work

In response to a lack of existing approaches specifically designed for testing database applications, we have proposed a framework for that purpose, and have designed and implemented a tool set to partially automate the process. This thesis focuses on several aspects: gathering information useful for testing from the DB schema and application queries, populating a database with meaningful data that satisfy constraints, and generating input data to be supplied to the application. Additional tools for checking the state after an operation has occurred and checking output have also been integrated into AGENDA, a comprehensive tool set that semi-automatically generates a database and test cases, and assists the tester in checking the results.

We have identified the issues that make testing database applications different from testing other types of software systems, explaining why existing software testing approaches may not be suitable for these applications. We have described the design of AGENDA and the functionality of each of its components: Agenda Parser, State Generator, Input Generator, State Validator, and Output Validator, focusing on the parsing and generation tools. We have demonstrated the feasibility of the approach with examples that were run on the system.

We have demonstrated AGENDA's ability to handle not-NULL, uniqueness, referential integrity constraints, and semantic constraints involving simple expressions. We have extended our approach to handle more complex semantic constraints, by extracting information from the query tree as well as the Abstract Syntax Tree. By extracting from the query tree, we have the means to traverse expressions in the query that may be complex and may contain many host variables. With feedback from the tester,

we can handle constraints that may not be explicitly included in the schema. We also use constraints that are *not* part of the schema to guide generation of tuples. For example, if there is no not-NULL constraint, a database entry should be included that has a NULL value. Running the application on an input that exercises this NULL value could expose a fault in the schema (i.e., that there should have been a not-NULL constraint) or a fault in the application's handling of NULL values. Similarly, if there is no uniqueness constraint, then there should be entries with duplicate values of the appropriate attribute included. This is accomplished by allowing the tester to select heuristics to guide the generation.

We are currently populating the database with either attribute values supplied by the tester or boundary values extracted from the DB schema and/or DB application, depending on the tester's choice. We investigated the interaction between the values used to populate the database and the values the tester enters as inputs to the application program. Though the tester may specify different heuristics for input generation than those specified for state generation, in order to fully utilize heuristics that guide the generation of the DB state, it is recommended that they also be used in the generation of the inputs. For example, if we choose the `nulls` heuristic to guide the State Generator, then nulls will appear in the DB state, but in order to ensure that nulls are also selected for test cases, then the `nulls` heuristic should also be chosen to guide the Input Generator.

6.1 Different levels of database application testing

This thesis initially focused on applications consisting of a single query and then addressed some of the issues involved in testing transactions consisting of multiple queries. Testing transactions is explored further in [17, 18]. To generate a test case for a transaction, the Input Generator instantiates each input parameter in the transaction. In doing so, it considers the integrity constraints and heuristics selected. A case study was presented, using a real application, the TPC-C benchmark, with a complex schema and complex transaction processing. This case study demonstrated the feasibility of extending our tool set to real applications. It also demonstrated the tool set's ability to expose errors, with the assistance of the tester in choosing appropriate data groups and heuristics. AGENDA's parsing and generation tools exposed more than half of the seeded errors. This result improved with the assistance of the checking tools, designed to expose semantic errors which violate postconditions in the resulting DB state and/or output [17, 8].

6.2 Different kinds of application domains

There are different kinds of application domains. Sometimes, we start with a particular database (student records, employee records, course information, etc) and then write different applications for it. AGENDA is useful for this application domain. Sometimes, we write an application which creates a database. For example, the TPC-C application begins with an empty database and populates the tables by running transactions. If we populate a database before running the TPC-C application, only considering the constraints in the schema and the heuristics selected by the user, we may begin with an unreachable DB state from the application's perspective. A different framework might be useful for this application domain, or perhaps we could extend the existing framework to allow the application to populate the DB, via inputs provided by the Input Generator.

6.3 Consistency issues

How do we generate an initial DB state that is consistent, not just with the constraints in the schema but additional constraints that are relevant for the application under test? How do we check the consistency of the resulting DB state, after the application executes a test case? The current generation of `type A` test cases, which takes into account relationships between attributes/parameters extracted from the schema, can be extended to handle additional relationships that make sense for the application under test, with some feedback from the tester. A feedback loop between the State Generator and Input Generator might also be helpful in generating an initial reachable DB state. Generation of `type B` test cases will be useful for testing the robustness of the application, by exercising it on test data it might not expect. For example, if a test case is produced for a given query such that its WHERE condition is not satisfied, then the application will be tested on whether the empty or NULL set returned by this query is handled or the application crashes.

6.4 Combinatorial issues

Generating test data based on all combinations of data groups or all templates could lead to a combinatorial explosion. We currently deal with this issue by providing a means to partition data values into groups and providing an `all groups` heuristic, for which it is sufficient that each data group is rep-

resented just once. However, an error may only be exposed when certain combinations of data groups appear together and interact. For example, a phone type of ISDN may only have a problem with a particular interface or switch market [42]. Latin squares, well known for their application to the design of statistical experiments, have also been applied to the combinatorial issue in software testing [32]. There are some limitations and assumptions imposed by the Latin squares method, but in practice, these can be relaxed [42]. Latin squares and/or a heuristic approach can achieve better coverage (pairwise or n-way) while avoiding an explosion in the amount of test data generated. Partitioning of data can be extended to allow hierarchies of data groups, so that if the combinations of groups at one level is too numerous, then the next higher level in the hierarchy can be examined, and so on, or the user can be prompted to combine certain groups in order to apply the Latin squares method more efficiently. The user could also be given the option to specify priorities for data groups, as in [10]. These functionalities can be incorporated into the AGENDA tool set as additional heuristics.

6.5 Automation issues

To what extent can database testing tools be automated, given reasonable assumptions? It is not assumed that there is a formal specification for the database application, since this is not realistic in practice. Therefore, the tool set cannot be fully automated. However, there is room for more automation. Currently, AGENDA extracts information from the database schema, sample-values files, and application queries, and uses this information to generate a DB state and input test cases, consistent with the integrity constraints in the schema and heuristics selected by the user. While AGENDA accepts any schema, it does not extract information from all possible constructs, as discussed in Section 5.2. Further analysis of the source code, consisting of the host language with embedded SQL queries, would provide additional information, that could be used to automatically partition or re-partition data groups, and moreover, to assist the current tools in generating consistent test data that is relevant for the application and checking the results. AGENDA's state checking and output checking tools are discussed further in [8]. Further analysis of the source code could also yield improvements in the efficiency of test data generation, such as pruning the database for applications that only use a part of the database.

The goal is to assist the database application developer or tester in a usable, useful way. The approach leverages the fact that the database schema is described formally in the Data Definition Language

of SQL, in order to ensure that the data generated satisfies the integrity constraints, and allows the user to provide additional information to guide the generation. Automating the testing process as much as possible is desirable, but not at the expense of usability. Users should not be burdened with having to describe their data and/or applications in yet another language, as is required by other approaches. With these goals in mind, future work involves further experimentation and extensions to increase the usefulness of the AGENDA tool set.

Appendix A

AGENDA DB Schema

Following are some relevant AGENDA DB tables. An example illustrating some of the values stored in these tables is provided in Sub-section 2.6.1.

Table `table_recs` keeps track of information about the tables in the Application DB, such as the name of each table, the number of attributes in each table, and information indicating a correct order to fill the tables.

```
create table table_recs (table_name varchar(25) primary key,
num_attributes int not null default 0, ref_type int not null default -1);
```

Table `attribute_recs` keeps track of information about the attributes in the Application DB, such as the name of each attribute, its data type, and which constraints (if any) exist on each attribute.

```
create table attribute_recs (table_name varchar(25), attr_name varchar(25),
data_type int, size int default 0,
is_primary bool not null default 'F', is_unique bool not null default 'F',
is_not_null bool not null default 'F',
is_check bool not null default 'F', is_default bool not null default 'F',
is_composite bool not null default 'F', is_boundary bool not null default 'F',
foreign_table varchar(25), foreign_attr varchar(25),
num_groups int not null default 0,
prob_or_freq varchar(4) not null default 'PROB',
```

```
primary key(table_name, attr_name),
foreign key(table_name) references table_recs,
check (foreign_table!='' OR foreign_attr=''),
check (foreign_attr!='' OR foreign_table=''),
check (prob_or_freq in ('PROB','FREQ')));
```

Table data_group_recs keeps track of information about the data groups associated with each attribute/parameter, such as the name of each data group, the attribute/parameter with which it is associated, the frequencies with which each group has been selected for the Application DB and test cases, and the number of values in each group.

```
create table data_group_recs (table_name varchar(25), attr_name varchar(25),
group_name varchar(25), num_values int not null default 0,
choice_percent float4 default 100.0,
choice_freq_in_db int default 0 not null,
choice_freq_in_tc int default 0 not null,
fk_choice_freq_in_db int default 0 not null,
choice_want_in_db int, choice_want_in_tc int,
index int not null default 0, random_index int not null default 0,
primary key(table_name, attr_name, group_name),
check(choice_percent>=0 and choice_percent<=100),
check(choice_freq_in_db>=0), check(choice_freq_in_tc>=0));
```

Table value_recs keeps track of information about the values associated with each attribute/parameter, such as the name of each value, the attribute/parameter with which it is associated, the group to which it belongs, and the frequencies with which each value has been selected for the Application DB and test cases.

```
create table value_recs (table_name varchar(25), attr_name varchar(25),
group_name varchar(25), value varchar(25),
val_freq_in_db int default 0 not null,
```

```
val_freq_in_tc int default 0 not null,
fk_val_freq_in_db int default 0 not null,
index int not null default 0, random_index int not null default 0,
check(val_freq_in_db>=0), check(val_freq_in_tc>=0),
check(index>=0), check(random_index>=0));
```

Table composite_recs keeps track of information about composite key constraints, such as a unique index for each composite constraint, the type of composite constraint (primary/unique/foreign), the table on which the constraint exists, the number of attributes involved in each composite constraint, and each attribute involved (one tuple per attribute, so that we can maintain bookkeeping for an arbitrary number of attributes).

```
create table composite_recs (composite_index int default -1,
composite_type varchar(2) not null default 'CP', table_name varchar(25),
num_attributes int default 2, attr varchar(25),
check(composite_type in ('CP', 'CU', 'CF')),
primary key(composite_index, composite_type, table_name, attr),
foreign key(table_name) references table_recs,
foreign key(table_name, attr) references attribute_recs);
```

Table boundary_recs keeps track of information about boundary values, such as each boundary value, the attribute with which it is associated, and the operator involved in the expression containing the boundary value.

```
create table boundary_values (table_name varchar(25),
attr_name varchar(25), value varchar(25), op varchar(2),
closed_or_open varchar(6) not null default 'CLOSED',
check (closed_or_open in ('CLOSED','OPEN')),
unique(table_name, attr_name, value),
foreign key(table_name) references table_recs,
foreign key(table_name, attr_name) references attribute_recs);
```

Table appl_query_info keeps track of information about application queries, such as a unique query ID, the query itself, the table and attribute which are potentially changed by the query, and the event (SELECT/UPDATE/INSERT/DELETE).

```
create table appl_query_info (query_ID int primary key,
query varchar(100), table_changed varchar(25), attr_changed varchar(25),
expected_num_rows int, event varchar(6) not null default 'NONE',
num_templates int, num_hvs int,
check (event in ('SELECT','INSERT','UPDATE','DELETE','NONE')));
```

Table appl_parameter keeps track of information about parameters in the application queries, such as the unique query ID associated with each parameter, the parameter name, the parameter type (INPUT/DIRECT, OUTPUT/DIRECT, INPUT/INDIRECT, OUTPUT/INDIRECT), the table and attribute associated with the parameter, whether the parameter is part of a query precondition or postcondition, and whether the parameter needs to be instantiated with a value by the Input Generator.

```
create table appl_parameter (query_ID int,
parameter varchar(25), param_type int default 0,
table_name varchar(25), attr_name varchar(25), pre_or_post int default 0,
tc_type int, tc_value varchar(25), gen_input int default 0,
foreign key(query_ID) references appl_query_info,
foreign key(table_name) references table_recs,
foreign key(table_name, attr_name) references attribute_recs,
check (param_type>=0 and param_type<=4),
check (pre_or_post>=0 and pre_or_post<=2));
```

Table template_info keeps track of information about test templates, such as the ID of the transaction that is relevant for each template, the template name, and information used by the checking tools to check the results of executing test cases for each template.

```
create table template_info (xact_ID int, name varchar(500),
expected_num_row int, num_row int, test_constraint varchar(200),
```

```
tableNum int, tableNames varchar(100), result int, test int,
primary key(xact_ID,name));
```

Table test_template keeps track of information about test cases, such as each test case value, the type of the value (INTEGER/STRING/MONEY/FLOAT/TIMESTAMP), the test case to which the value belongs, and the transaction ID, template, table, attribute and group associated with the value.

```
create table test_template (xact_ID int, name varchar(500),
table_name varchar(25), attr_name varchar(25), group_name varchar(25),
tc_type int default -1, tc_value varchar(25) default '',
tc_id int default 0, primary key(xact_ID, name, tc_id),
foreign key(xact_ID, name) references template_info);
```

Appendix B

TPC-C Schema

Following are relevant TPC-C tables, discussed further in Chapter 5.

```
CREATE TABLE warehouse (w_id INTEGER PRIMARY KEY, w_name CHAR(10),
w_street_1 CHAR(20), w_street_2 CHAR(20), w_city CHAR(20), w_state CHAR(2),
w_zip CHAR(9), w_tax FLOAT, w_ytd FLOAT);

CREATE TABLE district (d_id INTEGER, d_w_id INTEGER, d_name CHAR(10),
d_street_1 CHAR(20), d_street_2 CHAR(20), d_city CHAR(20), d_state CHAR(2),
d_zip CHAR(9), d_tax FLOAT, d_ytd FLOAT, d_next_o_id INTEGER,
PRIMARY KEY (d_id, d_w_id), FOREIGN KEY (d_w_id) REFERENCES warehouse (w_id));

CREATE TABLE customer (c_id INTEGER, c_d_id INTEGER, c_w_id INTEGER, c_first CHAR(16),
c_middle CHAR(2), c_last CHAR(16), c_street_1 CHAR(20), c_street_2 CHAR(20),
c_city CHAR(20), c_state CHAR(2), c_zip CHAR(9), c_phone CHAR(16), c_since TIMESTAMP,
c_credit CHAR(2), c_credit_lim FLOAT, c_discount FLOAT, c_balance FLOAT,
c_ytd_payment FLOAT, c_payment_cnt INTEGER, c_delivery_cnt INTEGER, c_data CHAR(250),
PRIMARY KEY (c_id, c_d_id, c_w_id),
FOREIGN KEY (c_d_id, c_w_id) REFERENCES district (d_id, d_w_id));
```

```
CREATE TABLE c_orders (o_id INTEGER, o_d_id INTEGER, o_w_id INTEGER, o_c_id INTEGER,
o_entry_d TIMESTAMP, o_carrier_id INTEGER, o_ol_cnt INTEGER, o_all_local INTEGER,
PRIMARY KEY (o_id, o_d_id, o_w_id),
FOREIGN KEY (o_c_id, o_d_id, o_w_id) REFERENCES customer (c_id, c_d_id, c_w_id));

CREATE TABLE history (h_c_id INTEGER, h_c_d_id INTEGER, h_c_w_id INTEGER,
h_d_id INTEGER, h_w_id INTEGER, h_date TIMESTAMP, h_amount FLOAT, h_data CHAR(24),
FOREIGN KEY (h_c_id, h_c_d_id, h_c_w_id) REFERENCES customer (c_id, c_d_id, c_w_id),
FOREIGN KEY (h_d_id, h_w_id) REFERENCES district (d_id, d_w_id));

CREATE TABLE new_order (no_o_id INTEGER, no_d_id INTEGER, no_w_id INTEGER,
PRIMARY KEY (no_o_id, no_d_id, no_w_id),
FOREIGN KEY (no_o_id, no_d_id, no_w_id) REFERENCES c_orders (o_id, o_d_id, o_w_id));

CREATE TABLE item (i_id INTEGER PRIMARY KEY, i_im_id INTEGER, i_name CHAR(24),
i_price FLOAT, i_data CHAR(50));

CREATE TABLE stock (s_i_id INTEGER, s_w_id INTEGER, s_quantity INTEGER,
s_dist_01 CHAR(24), s_dist_02 CHAR(24), s_dist_03 CHAR(24), s_dist_04 CHAR(24),
s_dist_05 CHAR(24), s_dist_06 CHAR(24), s_dist_07 CHAR(24), s_dist_08 CHAR(24),
s_dist_09 CHAR(24), s_dist_10 CHAR(24), s_ytd INTEGER, s_order_cnt INTEGER,
s_remote_cnt INTEGER, s_data CHAR(50), PRIMARY KEY (s_i_id, s_w_id),
FOREIGN KEY (s_w_id) REFERENCES warehouse (w_id),
FOREIGN KEY (s_i_id) REFERENCES item (i_id));


CREATE TABLE order_Line (ol_o_id INTEGER, ol_d_id INTEGER, ol_w_id INTEGER,
ol_number INTEGER, ol_i_id INTEGER, ol_supply_w_id INTEGER, ol_delivery_d TIMESTAMP,
ol_quantity INTEGER, ol_amount FLOAT, ol_dist_info CHAR(24),
```

```
PRIMARY KEY (ol_o_id, ol_number, ol_d_id, ol_w_id),
FOREIGN KEY (ol_o_id, ol_d_id, ol_w_id) REFERENCES c_orders (o_id, o_d_id, o_w_id),
FOREIGN KEY (ol_i_id, ol_supply_w_id) REFERENCES stock (s_i_id, s_w_id));
```

Bibliography

[1] R. Binder. Testing object-oriented software: a survey. *Journal of Software Testing, Verification, and Reliability*, 6:125–252, December 1996.

[2] Kelley C. Bourne. *Testing Client/Server Systems*. McGraw-Hill, New York, 1997.

[3] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 12–21. ACM Press, 1993.

[4] M. Y. Chan and S. C. Cheung. Testing database applications with SQL semantics. *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374, March 1999.

[5] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weyuker. A framework for testing database applications. *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 147–157, August 2000.

[6] David Chays and Yuetang Deng. Demonstration of AGENDA tool set for testing relational database applications. *International Conference on Software Engineering*, 2003.

[7] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. AGENDA: A test generator for relational database applications. Technical Report TR-CIS-2002-04, Department of Computer Science, Polytechnic University, 2002.

[8] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An AGENDA for testing relational database applications. *To appear in Journal of Software Testing, Verification, and Reliability*, 2004.

[9] Huo Yan Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. In black and white: An integrated approach to class level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, July 1998.

[10] T. Y. Chen, Pak-Lok Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. Technical Report TR-2003-01, Department of Computer Science and Information Systems, Hong Kong University, 2003.

[11] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] E. F. Codd. Further normalization of the database relational model. In R. Ruston, editor, *Database Systems, Courant Computer Science Series*, volume 6, pages 33–64. Prentice Hall, New Jersey, 1972.

[13] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, and G.C. Patton. Model-based testing in practice. In *International Conference on Software Engineering*. ACM Press, May 1999.

[14] C.J. Date and Hugh Darwen. *A Guide to the SQL Standard*. Addison-Wesley, New York, 1997.

[15] R. A. Davies, R. J. A. Beynon, and B. F. Jones. Automating the testing of databases. *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, June 2000.

[16] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

[17] Yuetang Deng, Phyllis G. Frankl, and David Chays. Testing database transaction consistency. Technical Report TR-CIS-2003-04, Department of Computer Science, Polytechnic University, 2003.

[18] Yuetang Deng, Phyllis G. Frankl, and Zhongqiang Chen. Testing database transaction concurrency. *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, 2003.

[19] Roong-Ko Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*. IEEE Computer Society Press, October 1991.

[20] Roongko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[21] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, Third Edition*. Addison-Wesley, New York, 2000.

[22] Phyllis G. Frankl and Roong-Ko Doong. Tools for testing object-oriented programs. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[23] Mohammed Ghriga and Phyllis G. Frankl. Adaptive testing of non-deterministic communications protocols. In *6th International Workshop on Protocol Test Systems*. IFIP, September 1993.

[24] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the 1998 International Symposium on Software Testing and Analysis*, pages 53–62. ACM Press, March 1998.

[25] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):243–252, June 1994.

[26] PostgreSQL Global Development Group. *PostgreSQL*. http://www.postgresql.org, 1999.

[27] Neelam Gupta, Aditya Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings Foundations of Software Engineering*. ACM Press, November 1998.

[28] Bogdan Korel. Automated software test generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[29] David Kung, Pei Hsia, and Jerry Gao. *Testing Object-Oriented Software*. IEEE Computer Society Press, 1998.

[30] P. Lewis, A. Bernstein, and M. Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2002.

[31] Norman R. Lyons. An automatic data generating system for data base simulation and testing. *Database*, 8(4):10 –13, 1977.

[32] Robert Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *ACM Computing Practices*, 28(10):1054–1058, October 1985.

[33] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[34] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2002.

[35] Don Slutz. Massive stochastic testing of SQL. In *VLDB*, pages 618–622. Morgan Kaufmann, August 1998.

[36] M.R. Stonebraker and L.A. Rowe. The design of postgres. In *Proc. ACM-SIGMOD International Conference on the Management of Data*. ACM Press, June 1986.

[37] Transaction Processing Performance Council. *TPC-Benchmark C*. http://www.tpc.org, 1998.

[38] W. T. Tsai, Dmitry Volovik, and Thomas F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering*, 16(3):316–324, March 1990.

[39] Elaine J. Weyuker, Tarak Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[40] Elaine J. Weyuker and Bingchiang Jeng. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3(3):254–270, July 1994.

[41] Lee J. White and Edward I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, May 1980.

[42] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. *IEEE ISSRE*, 1996.

[43] Jian Zhang, Chen Xu, and S. C. Cheung. Automatic generation of database instances for white-box testing. *Proceedings of 25th Annual International Computer Software and Applications Conference*, October 2001.