Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

AGENDA: A Test Generator for Relational Database Applications

David Chays Yuetang Deng Phyllis G. Frankl
Saikat Dan Filippos I. Vokolos Elaine J. Weyuker

Department of Computer and Information Science

Technical Report
TR-CIS-2002-04
8/08/2002

AGENDA: A Test Generator for Relational Database Applications

David Chays
Yuetang Deng
Phyllis G. Frankl
Saikat Dan
Polytechnic University [1]
6 Metrotech Center
Brooklyn, NY 11201
dchays@cis.poly.edu
ytdeng@photon.poly.edu
phyllis@morph.poly.edu

Filippos I. Vokolos
Drexel University
3141 Chestnut Street
Philadelphia, PA 19104
filip@mcs.drexel.edu

Elaine J. Weyuker
AT&T Labs - Research
Room E237
180 Park Ave.
Florham Park, NJ 07932
weyuker@research.att.com

August 8, 2002

[1]Supported in part by NSF Grants CCR-9870270 and CCR-9988354, AT&T Labs, and the New York State Office of Science, Technology, and Academic Research.

Abstract

Database systems play an important role in nearly every modern organization, yet relatively little research effort has focused on how to test them. This paper discusses issues arising in testing database systems, presents an approach to testing database applications, and describes AGENDA, a set of tools based on 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. The design and implementation of each component of the system are discussed.

Chapter 1

Introduction

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 paper 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 industrial organizations. This paper 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. Chapter 2 reviews relevant background and terminology. Chapter 3 discusses issues arising in testing database applications and describes our approach. Chapter 4 presents an overview of the tool set we built, called AGENDA, A (test) GENerator for Database Applications, and describes in more detail the design and implementation of each component of this system: a parsing tool to gather relevant information from the database schema and application, a tool to populate the database with values useful for testing the given application, a tool to generate test cases for the application, and tools to help the tester check the application's resulting database state and output. Chapter 5 compares our approach to the most closely-related commercial tools and research papers. Chapter 6 concludes with a summary of our contributions and directions for on-going work.

Chapter 2

Background and Terminology

2.1 Relational Databases and SQL

Relational databases are based on the relational data model, which views the data as a collection of relations [1, 2, 3]. 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.

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 [4].[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 $MSAccess^{TM}$.

Figure 3.1a 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.

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

Chapter 3

Issues in Testing DB Applications

In this section, we discuss several issues that arise in testing database applications. We illustrate these issues with the following simple hypothetical example, an application program that a telecommunications company might use to process customers' orders for new features (call-waiting, call-forwarding, etc.).

Assume the database includes a customer-feature table, with information about customers, including customer ID number, customer address, and features to which the customer subscribes, and a billing table, with customer ID numbers and information pertinent to issuing monthly bills, as well as other tables. The application program's specification is as follows:

> Input a customer's ID number and the name of the telephone feature to which the customer wishes to subscribe. If the ID number or the feature name is invalid, return code 0; otherwise, if the customer lives in an area in which that feature is available, and that feature is not incompatible with any other feature to which the customer already subscribes, add the selected feature to the customer's customer-feature record, update the billing table appropriately, and send a notice to the provisioning department (which will initiate the feature); return code 1. If the customer does not live in an area in which this feature is available, return code 2. If the feature is available in the customer's area, but it is incompatible with other features to which the customer subscribes, return code 3.

3.1 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 customer ID and the feature name, while the output is a numeric code between 0 and 3. This suggests that generating test cases is a matter of finding various "interesting" customer IDs and features, 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 (customer-ID, feature) 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 (customer-ID, feature) pair depends on whether the customer ID is valid, what features the customer already has, and where the customer lives. In addition to the customer-feature table and the billing table, a table containing zip codes (or other means of identifying areas in which a feature may be activated) and a table containing information about which features are compatible with one another will also have to be accessed, and are 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 a straw man that 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. 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, adding a new feature to a customer's account would give rise to several different test cases including:

- The customer already subscribes to that feature.
- The customer does not already subscribe to that feature, but it is not available in his or her area.
- The customer does not already subscribe to that feature, but it is not compatible with other selected features.
- The customer does not already subscribe to that feature, it is available in his or her area, and it is compatible with other selected features.
- The customer does not already subscribe to any features.

For each of these cases, the input value (customerID, feature) would be the same, but the expected outputs would be different. 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.

3.2 Selecting Interesting Database States

As demonstrated above, the expected results of a given test case depend not only on the customer ID and the feature selected, but also on the state of the database, including whether the customer has previously selected features that are incompatible with the new feature, and whether or not the feature is available in the customer's area. To save effort, it might be possible to create one original DB state representing several different customers, having a variety of different characteristics.

Populating the DB with meaningful values may involve the interplay between several tables. For example, there might be a table $T1$ describing incompatible features and another table $T2$ describing which features are available in which geographic areas. Assume that each row in $T1$ is a pair of (incompatible) features and that each row in $T2$ is a pair consisting of a zip code and a feature (that is available in that zip code). To create a good initial DB state for the above situation, one would need to include a record for a customer who currently subscribes to feature X and lives in a zip code Z where (X, Y) is in $T1$ and (Z, Y) is in $T2$, indicating that features X and Y are incompatible and that feature Y is available in Z. When the customer tries to sign up for feature Y, the application program should deny permission, since, although feature Y is available in the customer's area, it is not compatible with feature X, to which the customer already subscribes. Even more complicated rules indicating permissible feature combinations, as well as complex rules describing feature interactions, could exist.

3.3 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 customer been added

to the list of those customers who need the feature provisioned? Has an appropriate entry been made in the billing table? If the feature cannot be activated in the customer's area, or if the customer has conflicting features already selected, 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 customer 12345 lives in an area for which the "call forwarding on busy" feature is not available, have been included in the initial DB state in order to test the situation in which the feature cannot be added. This should be communicated to the tester, indicating that the pair (12345, call-forwarding-on-busy) should be included as an input, and that the resulting output code 2 with no associated change to the DB state should be expected.

3.4 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). However, 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. Furthermore, testing with data that are currently in use may be dangerous since running tests may corrupt the DB so that it no longer accurately reflects the real world. This would require that any changes made to the DB state would have to be undone. This may be difficult, particularly if the changes are extensive or if other "real" modifications to the DB are being performed concurrently with testing. Finally, there may be security or privacy constraints that prevent the application tester from seeing the live data. For all of these reasons, testers are frequently prohibited from testing using live data. One alternative that is sometimes used involves building an exact replica of a database and performing testing on this "real" data, albeit in the laboratory.

An alternative 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 consistent and interesting data. A *consistent database state* is one in which all constraints intended and specified by the database designer are obeyed [5]. If the DBMS enforces integrity constraints, one could attempt to fill it with arbitrary data, letting it reject data that don't satisfy the constraints. However, this is likely to be a very inefficient process. Instead, we should try to generate data that are known to satisfy the constraints and then populate the DB with it. Furthermore, the data should be selected 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

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 3.1: Examples: a) database schema definition and b) queries

correctly treated. In order to insure that the data are consistent, we can take advantage of the database schema, which describes the domains, the relations, and the constraints the database designer has explicitly specified. This information is expressed in a formal language, SQL's Data Definition Language (DDL), which makes it possible to automate much of the process.

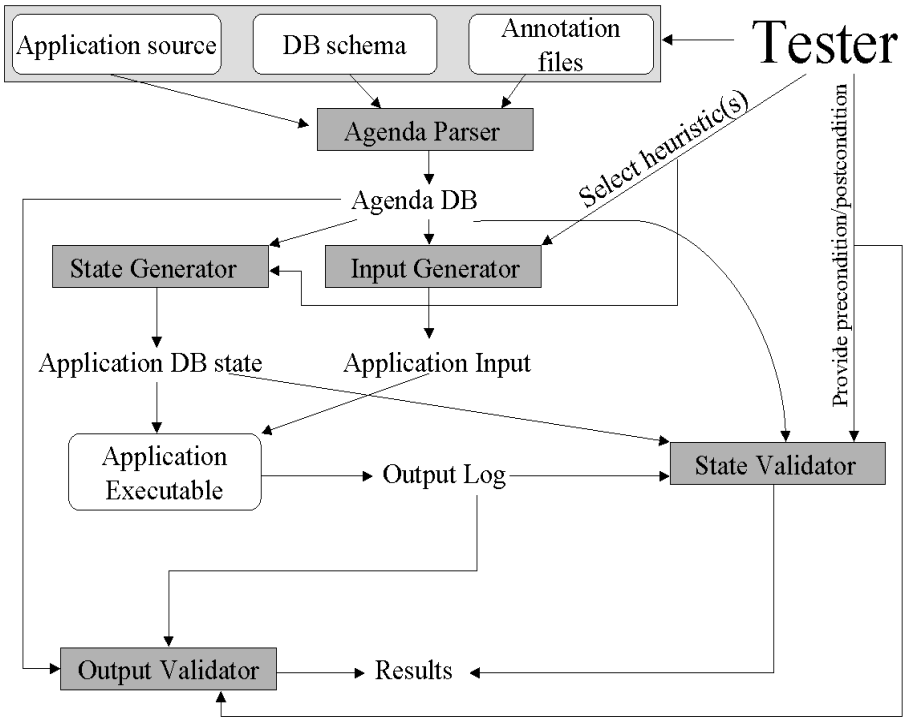

Figure 3.2: Architecture of the AGENDA tool set

Chapter 4

AGENDA Tool Set

4.1 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 3.2. AGENDA takes as input the database schema of the database on which the application runs; the application source code; and "annotation 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 and checks some aspects of correctness of the resulting database state and the application output. We currently assume that the application consists of a single SQL query. Ongoing work is relaxing this assumption.

The approach is loosely based on the Category-Partition method [6]: the user supplies suggested values for attributes, partitioned into groups, which we call *data groups.*[1] This data is provided in the annotation 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 subsection 4.3.3, below.

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.

Finally, AGENDA instantiates the templates with specific test values, executes the test

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

cases and checks that the outputs and new database state are consistent with the expected behavior indicated by the tester.

In the remainder of this section, the design and implementation of AGENDA are described in more detail, consisting 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 annotation 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 annotation 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.

In the following sections, we will use the schema and queries in Figure 3.1 to illustrate the operations of each component. 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.

4.2 Agenda Parsing Tool

4.2.1 Design of the Agenda Parser

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 [7] and now commercially supported by PostgreSQL [8]. PostgreSQL supports most of SQL-2 (along with additional features that are more object-oriented) and provides a well-documented open-source parser. The PostgreSQL parser creates a parse 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 3.1a are expressed using different syntactic constructs, leading to different parse sub-tree structures. For these reasons, we modified and extended the PostgreSQL parser so that as it parses the schema definition for the database underlying the application to be tested, we collect relevant information about the tables, attributes, and constraints associated with each attribute. An earlier version of the tool, described in [9], 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 [9], are avoided, and the information is more accessible to the other components of AGENDA, which are now database applications. Agenda Parser also extracts relevant information from the application source code and annotated files supplied by the tester. All of this information is stored in the Agenda DB.

The information in the Agenda DB is needed by the remaining four components to semi-automatically populate a database state, generate test cases for the application queries, and check the resulting database state and output. Some of the information that is stored in the Agenda DB is also stored in the DBMS's internal catalog tables. We could have designed the system so that the remaining components, the State Generator, Input Generator, State Validator, and Output Validator, queried these catalog tables. However, building and then querying a separate Agenda DB allows us to decouple the remaining components from the details of PostgreSQL. If we choose to base our tool set on a different DBMS, then the only tool that we would need to change is the Agenda Parser. Furthermore, the information in the catalog tables is not complete for our purposes. User information (such as data groups and sample values for attributes), which needs to be accessible to each tool of the system, is also stored in the Agenda DB. The Agenda DB is a repository of information for the tools to read and update, and in effect, communicate with one another. Thus, we are able to handle the interplay between the database state, the application inputs, and the expected results.

For the purpose of collecting information about the schema and application queries, the Agenda Parser is a modification of the PostgreSQL parser (DBMS parser). A separate component of the Agenda Parser updates the Agenda DB with information from the tester about data groups, data values, etc.

As the DBMS parses the schema, we extract information about tables, attributes, and constraints. The tool extracts constraint information, including not-NULL, uniqueness, and referential integrity constraints, as well as boundary values from semantic constraints. For the table `emp` in Figure 3.1a, the Agenda Parser extracts boundary values 6000.00 and 10000.00 along with their associated attributes (in this case, `salary` for both) from the parse tree and stores this information in the Agenda DB. For constraints on multiple attributes, which we refer to as *composite constraints*, the tool constructs data groups and value records for each of these composite attributes. Attributes that are parts of composites are marked as such in the Agenda DB so that they can be handled correctly during test generation. We are considering techniques to generate a representative subset of a cartesian product in order to control the explosion in the size of the number of attributes considered. One such technique, which has previously been used for controlling combinatorial explosions in test generation, is Latin Squares [10].

We extended the parser so that it can handle queries with *host variables*, variables in the host language that are used as parameters in SQL queries. As the DBMS parses an application's query, we extract information about input and output host variables. Again, we utilize the DBMS parser for this purpose because this is a much cleaner approach than writing our own parser from scratch.

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 3.1b 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 3.1b, Agenda Parser stores in the Agenda DB 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 (*direct* or *indirect*) 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). Further details on the operations of the Input Generator can be found in section 4.4.

Further details on how the Agenda DB, created by the Agenda Parser, is used by the other tools can be found in the upcoming sections.

The general procedure of the Agenda Parser is:

- Create the Agenda DB.
- Invoke the DBMS to fill the Agenda DB using information derived from the application DB schema.
- Back up the state of Agenda's tables so that if another application is tested that uses the same application DB schema, the Agenda Parser need not parse the application DB schema again.
- Fill the Agenda DB using information from the tester concerning data groups and values.
- Fill the Agenda DB using information derived from the application source code.
- Generate the test templates.

4.2.2 Implementation of the Agenda Parser

We faced some challenges in extracting information from queries. It was necessary to hack the lexer and the parser to handle the host variables. The DBMS does not handle a query if the host variables are not instantiated; it reports a parse error. On the other hand, if we instantiate the host variables ourselves (just for the purpose of extracting the above information from the query), then we'd have to write our own parser from scratch and/or reserve dummy values for host variables. We decided instead to modify the lexer so that a new token is recognized: the host variable, preceded by a colon. Next, we needed to modify the actions for parsing "SELECT...INTO" statements and add some new actions, because PostgreSQL expects a single *table* name in the into clause (whereas SQL92 uses "select...into" to represent selecting values into scalar variables of a host program). We also needed to traverse the structure in which PostgreSQL stores the target list (the attributes selected). We set flags when host variables are found so that certain actions are skipped when we are only interested in extracting and so that the meaning of statements that do not have host variables is preserved.

Information is extracted by traversing the query tree built by the DBMS. Thus, we are able to handle clauses with complicated expressions and many input and output host variables. Another advantage of pulling information from the query tree is that we can find boundary values in nodes tagged by PostgreSQL with a CONST (constant) label. For the `select` query in Figure 3.1b, Agenda Parser extracts the boundary values 7000 and 9000 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 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 4.3.2. 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.

Further details on how the Agenda DB, created by the Agenda Parser, will be used by the other tools can be found in the upcoming sections.

table Tables

Table	Number of attributes	Order to fill (refer to subsection 4.3.3)
dept	3	0
emp	5	1

table Attributes

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 4.1: Information in the Agenda DB after parsing the schema

4.2.3 Example

The purpose of this example is to show the kind of information stored in our internal database, the Agenda DB, by the Agenda Parser. The tools which use this information are described later.

With reference to the schema in Figure 3.1a and the `update` query in Figure 3.1b, 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 annotation file for `empno`, the sample values provided for attribute `empno` are partitioned into 3 groups: student, faculty, and administrator, as in Figure 4.2a.

The Agenda Parser stores information in the Agenda DB for this schema, as shown in Figure 4.1.

The Agenda Parser stores the following information in the Agenda DB for this query:

```
Unique query ID: 1

Host variables:
  Name       Type                          Attribute
  :rate      input/indirect association    salary
  :in_empno  input/direct association      empno

Attribute changed: salary
```

```
Precondition: empno = :in_empno
Postcondition: sal = sal * :rate

3 data groups for attribute emp:
  student, faculty, administrator

3 test templates for this query:
  template_emp_empno_student,
  template_emp_empno_faculty,
  template_emp_empno_administrator

2 boundary values for attribute salary:
  5000.00, 10000.00
```

4.3 State Generation Tool

4.3.1 Design of the State Generator

Here, we describe the design of the second component of the tool set, the tool to populate the database. The tool has been designed to allow flexibility so that it can be used for experimentation with different test generation strategies.

The tool inputs a database schema definition, along with some additional information from the tester, retrieved from the Agenda DB, and outputs a consistent database state, in which all constraints specified in the schema and by the tester are obeyed. It incorporates various heuristics to aid in generating useful DB states (i.e., states deemed likely to expose application faults).

For example, the tester might provide the file shown in Figure 4.2a (among others) for testing the application whose schema and queries are shown in Figure 3.1. 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 below.

We have deliberately chosen this semi-automated generation approach, rather than attempting to automatically generate attribute values satisfying the domain 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 are associated with some domains (e.g. integers within a given subrange) and testers supply possible values for other domains.

4.3.2 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

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

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 3.1, 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 [11]. 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 [12].

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 4.3. 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 4.3.

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 that might fail 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.

4.3.3 Implementation details of the State Generator

An input file may contain only data values, or it may contain a mix of data values and annotations that specify how the values should be selected by the tool. Currently the tool supports five different annotations: `choice_name`, `choice_prob`, `choice_freq`, `null_prob`, and `null_freq`. The annotation `choice_name` partitions the data values in different data groups. The annotation `choice_prob` specifies the probability for selecting a value from the list of values that follows, while the annotation `choice_freq` specifies the frequency with which values should be selected.[2]

Once the tester's input files have been used to fill the `data groups` and `values` records of the Agenda DB, the tool can begin to generate tuples for insertion into the database tables. For a given table and a given attribute, the `choice_prob` or `choice_freq` annotation (if present) is used to guide selection of a data group or of NULL. After selecting a data group, the tool selects a value from the group, obeying the constraints, as described below, and marks that the value has been used.

[2]An annotation `choice_prob:50` tells the tool to randomly select a data group, with this data group having a probability of selection of 0.5; thus, it is *likely* that *roughly* 50% of the selected values will come from this data group. An annotation `choice_freq:50` tells the tool to select an element of this data group *exactly* 50% of the time, if it is possible to do so without violating constraints.

Handling not-NULL constraints is straightforward. If there is a uniqueness constraint, the `used` field in the value record is marked when that value is selected; the field is checked before selecting values to avoid selecting the same value more than once.

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 tool 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.

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 ≥ 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. Tables T and T' cannot reference each other on the same column, because any insertion would violate the foreign key constraint; this would mean that column in T is an exact duplicate of the corresponding column in T'; if the user really wanted that, he/she would have to use another means, such as a *trigger*. 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 [4]. 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 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 3.1a, the output of the topological sort based algorithm is shown in the column "Order to fill" of the first Agenda DB table in Figure 4.1, indicating that table `dept` must be filled before table `emp`.

4.3.4 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 3.1, 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.

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.

4.3.5 Example

In this section we present an example illustrating a database state produced by the tool for the Department-Employee schema shown in Figure 3.1a. Input files supplied by the tester are shown in Figure 4.4. The file for the `salary` attribute, automatically generated as described in subsection 4.3.2, is shown in Figure 4.3.

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 4.5. These have been obtained by executing the `insert` statements generated by the tool (see Figure 4.2b) and then outputting the resulting database state. 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). 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.

4.4 Input Generation Tool

4.4.1 Design of the Input Generator

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 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 3.1b.

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`, `duplicates in the test cases`, `nulls`, `values not in the application DB`, `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 3.1, 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.

There are two types of `duplicates` that the tester may want to select. 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 `duplicates from the application DB state`, 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 from the application DB state` 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. By selecting `duplicates in the test cases`, the

tester indicates that some of the test cases should include the same value for a specific input parameter. For example, assuming the three tuples mentioned above, if 222 is selected for two different test cases, then that would count as a `duplicate in the test cases`, even though 222 appears only once in the application DB.

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 `values not in the application DB`, then values that were not selected by the State Generator for inclusion in the application DB state are selected by the Input Generator for inclusion in the test cases. The Input Generator finds these values by checking the appropriate `used` field in the Agenda DB.

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. Test templates are discussed in more detail in the example below.

4.4.2 Implementation of the Input Generator

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 4.5, 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.

4.4.3 Example

The Input Generator generates test cases by instantiating values for the application query's parameters, or input host variables. The `update` query in Figure 3.1b 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 4.4. 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 4.6. 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 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).

4.5 State Validation Tool

After a test case is executed, the output consists of the output returned to the user, along with the modified DB state. Unless there is a formal specification of the intended behavior of the application, including a specification of how the database state will be changed, it is not possible to fully automate checking of the application's results. However, it is often realistic to expect testers to know something about expected state changes, including which tables should have been modified by the application, and what the resulting changes to the tables should be. The tool allows the tester to specify preconditions, postconditions, and relationships between old and new values. This information, combined with test case information, is used to generate integrity constraints for the log tables. Further details and examples are provided later.

4.5.1 Automatic Logging of Changes in the Application Tables

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 [3]. We use triggers as the means of capturing changes in tables in our tool in the following ways:

- Modify the schema so that for each table, there is an additional log table that records all modifications made to the table when the application program is executed;
- Add triggers that put entries into the appropriate log table in response to each insert, modify, or delete operation performed on the base tables by the application;
- Query the log tables to obtain information about the changes made to the database by the application.

We define a trigger to capture changes to a specific table. Whenever a change is made to the table, the trigger is automatically executed, leading to appropriate rows being inserted into a predefined log table associated with each table. The schema of a log table is very similar to that of the application table. For the table `emp` in Figure 3.1a, the tool begins by defining its log table in SQL as follows:

```
CREATE TABLE emp_log (event INT, empno_old INT, ename_old CHAR(25), salary_old MONEY,
bonus_old MONEY, deptno_old INT, empno_new INT, ename_new CHAR(25), salary_new MONEY,
bonus_new MONEY, deptno_new INT)
```

The first attribute, `event`, indicates the kind of event (insert, update, delete or select). The remaining fields are used to store values of the attributes of the base table (in this case, table `emp`) before and after modifying the base table. If the event is SELECT, then the table is not changed, so we only store the old values in the `emp_log` table.

In PostgreSQL, many things that can be done using triggers can also be implemented using active rules. An *active rule* is a means of specifying actions that take place automatically in response to some event. Whereas a trigger fires once per row, an active rule can handle many rows at once by manipulating the parse tree or generating a new one. So if many rows are affected in one statement, an active rule issuing just one extra query is more efficient than a trigger [8].

We therefore define three rules, corresponding to updating, inserting, or deleting:

```
CREATE RULE emp_update AS ON UPDATE TO emp DO INSERT INTO emp_log VALUES
(1, old.empno, old.ename, old.salary, old.bonus, old.deptno, new.empno,
new.ename, new.salary, new.bonus, new.deptno)

CREATE RULE emp_insert AS ON INSERT TO emp DO INSERT INTO emp_log VALUES
(2, 0, '', 0, 0, 0, new.empno, new.ename, new.salary, new.bonus, new.deptno);

CREATE RULE emp_delete AS ON DELETE TO emp DO INSERT INTO emp_log VALUES
(3, old.empno, old.ename, old.salary, old.bonus, old.deptno, 0, '', 0, 0, 0);
```

Rule `emp_update` fires once for all the rows in table `emp` that are updated. It inserts one row into table `emp_log` each time a row in table `emp` is updated. First, it sets `event` equal to 1 indicating that an "update" operation is being performed, and then it writes the values before updates (`old.empno`, `old.ename`, `old.salary`, `old.bonus`,`old.deptno`) into specified attributes (`empno_old`, `ename_old`, `salary_old`, `bonus_old`, `deptno_old`). It next writes the values after updates (`new.empno`, `new.ename`, `new.salary`, `new.bonus`, `new.deptno`) into specified attributes (`empno_new`, `ename_new`, `salary_new`, `bonus_new`, `deptno_new`). Rules `emp_insert` and `emp_delete` work similarly for insertions and deletions, respectively, except that `emp_insert` writes empty values (0 or null string `''` depending on the attribute) into old attribute states and `emp_delete` writes empty values into new attribute states. The `event` is set to 2 or 3, indicating that an "insert" or "delete" has been performed, respectively.

4.5.2 Overhead Estimation of the Rules

In order to investigate how the addition of these rules affects the system performance, we measured the execution time during various operations, with and without the logging rules. We ran 10,000 insertions, 10,000 updates, and 10,000 deletions, each on three tables T5, T20, and T100, where T5 has 5 integer attributes, T20 has 20 integer attributes, and T100 has 100 integer attributes. We treated all of the 10,000 operations as both a single transaction, and as 10,000 transactions. The time to execute these transactions is shown in Figure 4.7. The first two columns show the total cost in seconds without rules and with rules, respectively. The last column shows the overhead due to using rules, defined as the difference between execution time with rules and without rules divided by the execution time without rules, multiplied by 100%.

With one exception (T5 versus T20, insertion, 10000 transactions), the percentage overhead increased as the number of attributes increased. When the modifications were done as 10,000 separate transactions, the overhead ranged from 24% to 92% for all operations on the 3 tables. In general, if the operation only changed a small portion of the database (such as updating 1 row of a table with 10000 rows), the cost of using a rule was relatively small. When the 10,000 modifications were done in one transaction, however, the overhead ranged from 38% to 400%. If the operation changed a large portion of the database, the cost of using a rule may be high. Given that the overhead associated with the use of a rule will only be incurred during testing, we believe the amount of overhead that we have seen is reasonable.

Our technique also entails overhead in terms of space. The number of rows in the log table is the same as the number of rows changed in the associated table. This is the asymptotically optimal space solution to keep track of the exact changes. If it is impractical to store all of the changes, space could be saved by saving only the initial values of logged attributes in the log tables and/or by hashing attribute values and storing those, then comparing them to hashes of the new values.

The State Validator tool constructs the log tables and rules automatically from the user's database schema. First, the Agenda Parser tool parses and extracts all of the necessary information from the user's database schema, and stores this infomation in the Agenda tables. Then, the State Validator automatically generates the log tables and rules for updating the log tables by querying these Agenda tables. By adding these log tables and rules to the user's table, we have a tool that automatically shows the tester exactly how the database state has changed and gives statistical information about the change, including how many rows were updated and what changes occurred.

4.5.3 Checking the State Change by Means of Database Constraints

To check the DB state, we need to consider the following issues:

- Checking that tables that should not have changed didn't change and checking that tables that should have changed did change.

- Checking that tables changed in the correct way, according to constraints generated by the tool and supplied by the tester.
- Checking that the new-state satisfies the relevant constraints, including those specified by the tester as well as those defined in the schema and application.

The automatic checking of integrity constraints by a DBMS is one of the more powerful features of SQL [13]. Various semantic integrity constraints are supported in SQL standards. We find check/assertion constraints are very suitable to validate the changes.

In SQL standard SQL-92/SQL-99, the `check constraint` defines a general integrity constraint that must hold for each row of a table. `Assertion` defines a named, general integrity constraint that may refer to more than one table. Embedded SQL refers to SQL statements placed within an application program. A parameterized query is a SQL query with host variables. In this paper, we focus on transactions which consist of only a single query.

ditions and/or postconditions. lly changed/affected; check whether it equals the expected number of rows to make sure that nothing else changed; if not, report an error.

The general procedure of the State Validator is as follows:

- After the Agenda Parser parses the application schema, the State Validator generates a log table for each application table and generates rules to update the log tables automatically.
- After the Agenda Parser parses the application query and generates all the test templates based on data groups information of input host variables, the State Validator gives the tester the option to provide some precondition/postcondition for each test template.
- After the State Generator generates a consistent application DB state, and the Input Generator generates a test case for a specific test template, the State Validator combines the test case information (which becomes part of the precondition) and precondition/postcondition into an integrity constraint. The State Validator records the number of rows satisfied by the precondition.
- After execution of the test case, the State Validator applies the integrity constraint to the log table to check the resulting database state changes. The tool checks if the number of rows in the log table is equal to the expected number of rows.
- After completion of the test case, the State Validator drops the integrity constraint and clears the log tables.

4.5.4 Example

Suppose we have the schema in Figure 3.1a and the `update` query in Figure 3.1b and the Agenda Parser extracts the information as described in subsection 4.2.3.

The tool asks the tester to specify the expected result for each template. In this case, only one input host variable, with 3 data groups, is considered, so there are 3 templates.

Suppose that the tester inputs "sal=sal*1.1" for the student group, "sal=sal*1.2" for the faculty group, and "sal=sal*1.3" for the administrator group. Then, the tool generates 3 template check constraints and one template query:

```
ALTER TABLE emp_log ADD CONSTRAINT ic_emp_empno_student
  CHECK (sal_new = sal_old * 1.1 AND empno_old = :in_empno)
ALTER TABLE emp_log ADD CONSTRAINT ic_emp_empno_faculty
  CHECK (sal_new = sal_old * 1.2 AND empno_old = :in_empno)
ALTER TABLE emp_log ADD CONSTRAINT ic_emp_empno_administrator
  CHECK (sal_new = sal_old * 1.3 AND empno_old = :in_empno)
SELECT COUNT(*) FROM emp WHERE empno = :in_empno
```

Suppose that sample value 111 in the student group has been inserted in the application DB by the State Generator tool, and 111 is chosen for a test case by the Input Generator tool. As the test case 111 belongs to the student group, the tool instantiates the template constraint and template query as follows:

```
ALTER TABLE emp_log ADD CONSTRAINT ic_emp_empno_student
  CHECK (sal_new = sal_old * 1.1 AND empno_old = 111)
SELECT COUNT(*) FROM emp WHERE empno = 111
```

The tool applies the constraint to the `emp_log` table, and stores the result of the query as expected_number_rows in Agenda's DB.

After running the test case, if no constraint is violated, the tool submits another query on table `emp_log`:

```
SELECT COUNT(*) FROM emp_log
```

The tool checks if the result of this query equals the value of expected_number_rows; if not, it reports an error.

Finally, after the tool completes the test for the test case, it drops the constraint.

```
ALTER TABLE emp_log DROP CONSTRAINT ic_emp_empno_student
```

Suppose that sample value 555 in the faculty group is not in the application DB but 555 is chosen for a test case by the Input Generator tool, according to the heuristic `values not in application DB`.

As the test case 555 belongs to the faculty group, the tool instantiates the template constraint and template query as follows:

```
ALTER TABLE emp_log ADD CONSTRAINT ic_emp_empno_faculty
  CHECK (sal_new = sal_old * 1.2 AND empno_old = 555)
SELECT COUNT(*) FROM emp WHERE empno = 555
```

Then the tool applies the constraint to the `emp_log` table, and stores the result of the query as expected_number_rows in Agenda's DB. In this case, since no rows satisfy the precondition of the application query, the expected number of rows is 0.

After running the test case, if no constraint is violated, the tool submits another query on table `emp_log`:

```
SELECT COUNT(*) FROM emp_log
```

Finally, the tool checks if the result of this query equals the value of expected_number_rows. In this example, if the log table is not empty, we know that some row changed that shouldn't have changed.

4.6 Output Validation Tool

The output returned to the user may contain more than one row in the application tables so we need to store the result in the log table and check whether all rows in the log table satisfy the constraints specified by the tester. The Output Validator is similar to the State Validator tool. When the application query is a `select` statement, it is handled by the Output Validator; otherwise, it is handled by the State Validator. Since most DBMSs do not support the creation of a trigger/rule that acts upon a `select` statement, we cannot use a trigger/rule to capture those rows affected by a `select` query. So the Output Validator submits a query on the application tables and saves affected rows in the log table. This query is generated from the application query. This is the main difference between the State Validator and Output Validator. The other parts of the Output Validator are the same as those of the State Validator.

The general procedure of the Output Validator is as follows:

- After the log tables and test templates have been generated, the Output Validator uses the log tables to store the results of the application's `select` query.
- After the State Generator generates a consistent application DB state, and the Input Generator generates a test case for a specific test template, the Output Validator combines the test case information (which becomes part of the precondition) and precondition/postcondition into an integrity constraint.
- The Output Validator submits a query based on the application query to record those rows that are executed by the application query and saves the output of this query in the log table.
- After execution of the test case, the Output Validator applies the integrity constraint to the log table to check the resulting output.
- After completion of the test case, the Output Validator drops the integrity constraint and clears the log tables.

```
--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 4.3: File generated for `salary` attribute

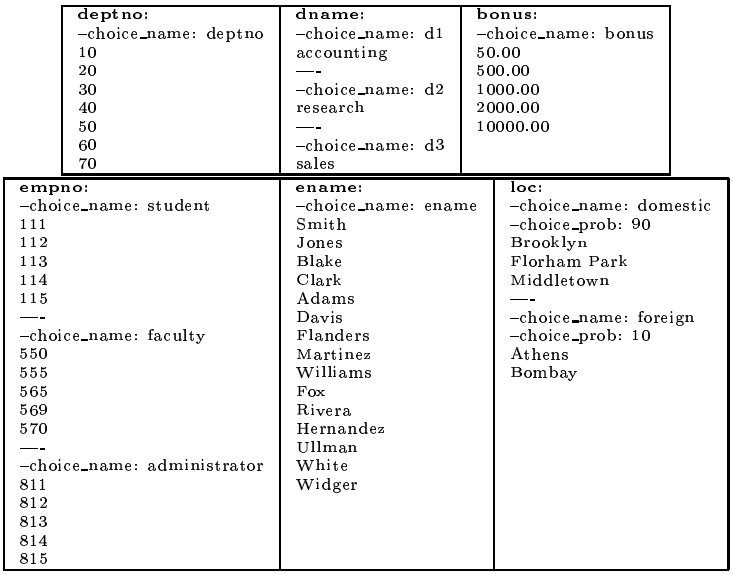

Figure 4.4: Input files for Department-Employee database

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 4.5: A database state produced by the tool

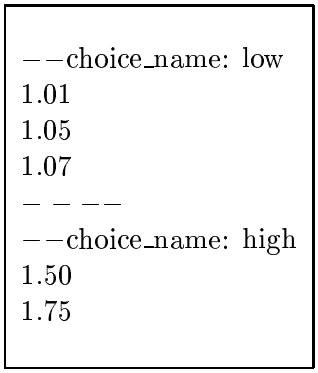

Figure 4.6: Input file for `rate` parameter of `update` query in Figure 3.1b

	time without rule	**time with rule**	**percentage overhead**
T5 / 1 transaction			
insert	87	120	38%
update	153	227	48%
delete	114	183	33%
T20 / 1 transaction			
insert	140	248	77%
update	173	337	95%
delete	123	252	105%
T100 / 1 transaction			
insert	350	748	114%
update	459	1114	143%
delete	133	665	400%
T5 / 10000 transactions			
insert	927	1214	31%
update	1277	1588	24%
delete	697	1079	55%
T20 / 10000 transactions			
insert	1016	1320	30%
update	1299	1752	35%
delete	701	1098	57%
T100 / 10000 transactions			
insert	1366	2058	51%
update	1645	2612	59%
delete	830	1591	92%

Figure 4.7: Time (in seconds) for executing transactions with and without rules

Chapter 5

Related Work

There is little work in the software testing research literature on testing techniques targeted specifically toward database applications. Davies, Beynon, and Jones [14] populate a database with a prototype that requires the user to provide validation rules to specify constraints on attributes. Tsai, Volovik, and Keefe [15] automatically generate test cases from relational algebra queries. There are also some practical guidelines for practitioners, as in [16].

Several techniques and tools have been proposed for automated or semi-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 [17, 18, 19, 20].

The approach of Chan and Cheung [21] 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 Generator, 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. Zhang, Xu, and Cheung [22] generate a set of constraints which collectively represent a property against which the program is tested. Database instances for program testing can be derived by solving the set of constraints using existing constraint solvers.

Our technique is more closely related to techniques used for specification-based test generation, e.g. [23]. The partially automated Category-Partition technique [6], introduced by Ostrand and Balcer, and described above, 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 [24]. 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.

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 [25, 26]. Another aspect of performance testing is addressed

by Slutz [27], 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. [28] 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 [29]. 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 some special purpose language. Another commercial tool, DataFactory (http://www.quest.com/datafactory), provides the tester with options to insert existing values, sequential values, random values, or constant values as well as an option to choose null probability, but only with respect to an entire column. 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 helps the tester to determine how the database state changes when the application is executed; the related approaches do not provide support for this important aspect of testing.

Chapter 6

Conclusions

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 paper focuses on several aspects: populating a database with meaningful data that satisfy constraints, incorporating boundary values extracted from semantic constraints in the DB application and DB schema, generating input data to be supplied to the application, checking the state after an operation has occurred, and checking output. These are components of a comprehensive tool set, AGENDA, 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. 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 parse tree. In our initial approach, we extracted from the parse tree; now, we are also extracting from the query tree, so 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.

A major issue for general software testing techniques is the determination of whether or not the output produced as the result of running a test case is correct. As discussed in the paper, this issue is especially problematic for database applications since the outputs include the entire database state which will generally be large and complex. We therefore developed ways of validating the output of test cases as they are executed. This includes checking that parts of the database that should have remained unchanged by an operation have indeed remained unchanged, and that those that should have changed did so in appropriate ways. We have designed and implemented the State Validator and Output Validator. The State Validator uses active database techniques (*trigger/rule*) to capture the changes in the application DB, and uses database integrity constraint techniques to check that the application DB state changed in the right way. It automatically shows the tester exactly how the database state has changed and gives statistical information about the change. The Output Validator stores the tuples which are satisfied by the application query and constraints in the log table. It uses the integrity constraint techniques to check that the precondition and postcondition hold for the output returned by the application.

We believe that database application testing contains three levels: the query level, the transaction level, and the transaction concurrency level. The query level considers problems associated with individual queries. The transaction level considers problems associated with groups of related queries. The transaction concurrency level deals with problems associated with simultaneously running transactions.

Currently, our system operates on the query level. We plan to add more features to our existing system on the query level, and extend our tool set to the transaction level and transaction concurrency level. Empirical evaluation, via a large scale experiment or case study, can then be performed.

Acknowledgments

Thanks to Jayant Haritsa, Alex Delis, Jeff Damens, Thomas Lockhart, Gleb Naumovich, Torsten Suel, Vinay Kanitkar, and the anonymous referees for providing useful suggestions.

Bibliography

[1] E. F. Codd, "A relational model for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[2] E. F. Codd, "Further normalization of the database relational model," *Communications of the ACM*, 1972.

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

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

[5] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, *Database System Implementation*, Prentice Hall, New Jersey, 2000.

[6] Thomas J. Ostrand and Marc J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, June 1988.

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

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

[9] 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*, pp. 147–157, Aug. 2000.

[10] Robert Mandl, "Orthogonal latin squares: An application of experiment design to compiler testing," *ACM Computing Practices*, vol. 28, no. 10, pp. 1054–1058, Oct. 1985.

[11] Lee J. White and Edward I. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 247–257, May 1980.

[12] Elaine J. Weyuker and Bingchiang Jeng, "A simplified domain-testing strategy," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 3, pp. 254–270, July 1994.

[13] Philip M. Lewis, Arthur Bernstein, and Michael Kifer, *Databases and Transaction Processing: An Application-Oriented Approach*, Addison-Wesley, New York, 2002.

[14] 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.

[15] 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*, vol. 16, no. 3, pp. 316–324, March 1990.

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

[17] R. A. DeMillo and A. J. Offutt, "Experimental results from an automatic test case generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109–127, April 1993.

[18] Bogdan Korel, "Automated software test generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, Aug. 1990.

[19] 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*. Mar. 1998, pp. 53–62, ACM Press.

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

[21] 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*, pp. 363–374, March 1999.

[22] 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.

[23] Elaine J. Weyuker, Tarak Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353–363, May 1994.

[24] 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*. May 1999, ACM Press.

[25] Transaction Processing Performance Council, *TPC-Benchmark C*, 1998.

[26] 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*. 1993, pp. 12–21, ACM Press.

[27] Don Slutz, "Massive stochastic testing of SQL," in *VLDB*. Aug. 1998, pp. 618–622, Morgan Kaufmann.

[28] 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)*, vol. 23, no. 2, pp. 243–252, June 1994.

[29] Norman R. Lyons, "An automatic data generating system for data base simulation and testing," *Database*, vol. 8, no. 4, pp. 10 –13, 1977.