

An AGENDA for testing relational database applications



David Chays¹, Yuetang Deng², Phyllis G. Frankl^{2,*}, Saikat Dan²,
Filippos I. Vokolos³ and Elaine J. Weyuker⁴

¹*Department of Mathematics and Computer Science, Adelphi University, 1 South Avenue, Garden City, New York, NY 11530, U.S.A.*

²*Department of Computer Science, Polytechnic University, 6 Metrotech Center, Brooklyn, New York, NY 11201, U.S.A.*

³*Department of Computer Science, 3141 Chestnut Street, Drexel University, Philadelphia, PA 19104, U.S.A.*

⁴*AT&T Labs—Research, Room C213, 180 Park Avenue, Florham Park, NJ 07932, U.S.A.*

SUMMARY

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 to facilitate the use of this approach. In testing such applications, the state of the database before and after the user's operation plays an important role, along with the user's input and the system output. A framework for testing database applications is introduced. A complete tool set, based on this framework, has been prototyped. The components of this system are a parsing tool that gathers relevant information from the database schema and application, a tool that populates the database with meaningful data that satisfy database constraints, a tool that generates test cases for the application, a tool that checks the resulting database state after operations are performed by a database application, and a tool that assists the tester in checking the database application's output. The design and implementation of each component of the system are discussed. The prototype described here is limited to applications consisting of a single SQL query. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: database; software testing; test data

*Correspondence to: Phyllis G. Frankl, Department of Computer Science, Polytechnic University, 6 Metrotech Center, Brooklyn, New York, NY 11201, U.S.A.

†E-mail: phyllis@morph.poly.edu

Contract/grant sponsor: NSF; contract/grant number: CCR-9870270, CCR-9988354

Contract/grant sponsor: AT&T Labs

Contract/grant sponsor: New York State Office of Science, Technology, and Academic Research



1. INTRODUCTION

Databases 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 ensuring 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 assess the quality of the database application programs. To address this need, a systematic, partially automatable approach to testing database applications was developed together with a tool set based on this approach.

There are many aspects of 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. 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, database 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 database 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. Attention is restricted to relational databases. Definitions of relevant



relational database terminology can be found in a database text book, such as that of Elmasri and Navathe [1]. Section 2 discusses issues arising in testing database applications and describes the approach. Section 3 presents an overview of the tool set 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. The AGENDA prototype described in this paper is limited to applications consisting of a single SQL query. Additional limitations are described in Section 3.7.3. Ongoing work involves relaxing these limitations. Section 4 compares the new approach to the most closely related commercial tools and research papers. Section 5 concludes with a summary of the contributions and directions for additional work.

2. ISSUES IN TESTING DATABASE APPLICATIONS

This section discusses several issues that arise in testing database applications. These issues are discussed further elsewhere by Chays *et al.* [2].

The techniques and tools considered in this paper are targeted toward database applications written in *SQL*, a standardized language for defining and manipulating relational databases [3][‡]. *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 that 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 *MS Access*TM.

Figure 1(a) 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* ≥ 6000.00) AND (*salary* ≤ 10000.00))' indicates that all values for the attribute *salary* must be between 6000.00 and 10000.00 inclusive.

2.1. The role of the database state

A database application, like any other program, can be viewed as computing a partial function (or, more generally, a relation) from an input space *I* to an output space *O*, although as discussed below, care

[‡]*SQL* will be used to refer to the 1992 standard, also known as *SQL2* or *SQL-92*, and dialects thereof, unless otherwise noted.



(a)

```
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)

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

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

Figure 1. Examples: (a) database schema definition in SQL and (b) queries.

must be taken regarding the role of the database state. Its specification can be expressed as a function (or relation) from I to O . Thus, a database application can be tested 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. For many software systems, it is non-trivial to determine whether or not the output produced in response to an input is correct. This problem is exacerbated for database applications, since knowledge of expected and actual values of the database state after executing the application is also needed in order to determine whether the application behaved correctly.

There are several possible approaches to dealing with the role of the database state. One could ignore the state, and simply view the application as a relation between the user’s inputs and the software’s outputs. This is obviously unsuitable, since such a mapping would be non-deterministic, making it essentially impossible to validate test results or to re-execute test cases. Alternatively, one could 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. Thirdly, one could treat the database state as part of both the input and output spaces. If the environment is modeled in sufficient detail, then the second and third approaches 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.

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. Note that populating the database with meaningful values may



involve the interplay between several tables. 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.

Note that there is some interplay between the database states before and after running the application and the inputs supplied by the user. If the database has been populated with particular values in order to test a particular aspect of the application's behaviour, then inputs must be given to the application that force access to the relevant tables, rows, and attributes.

2.2. Populate database with live or synthetic data?

One approach to obtaining a database state is simply to 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, discussed by Chays *et al.* [2].

An alternative approach is to generate data specifically for the purpose of testing and to run the tests in an isolated environment. Since the database 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. It is not sufficient to populate the database 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 [4]. If the DBMS enforces integrity constraints, one could attempt to fill it with arbitrary data, letting it reject data that do not satisfy the constraints. However, this is likely to be a very inefficient process. Instead, one should try to generate data that are known to satisfy the constraints and then populate the database 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 correctly treated. In order to ensure that the data are consistent, one 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.

3. AGENDA TOOL SET

This section describes in more detail the prototype tool set called AGENDA, A (test) GENerator for Database Applications.

3.1. System overview

The AGENDA tool set was developed to address the issues described above. The AGENDA architecture is shown in Figure 2.

AGENDA takes as input the database schema of the database on which the application runs; the application source code; and 'sample-values files', containing some suggested values for attributes. The user interactively selects test heuristics and provides information about expected behaviour

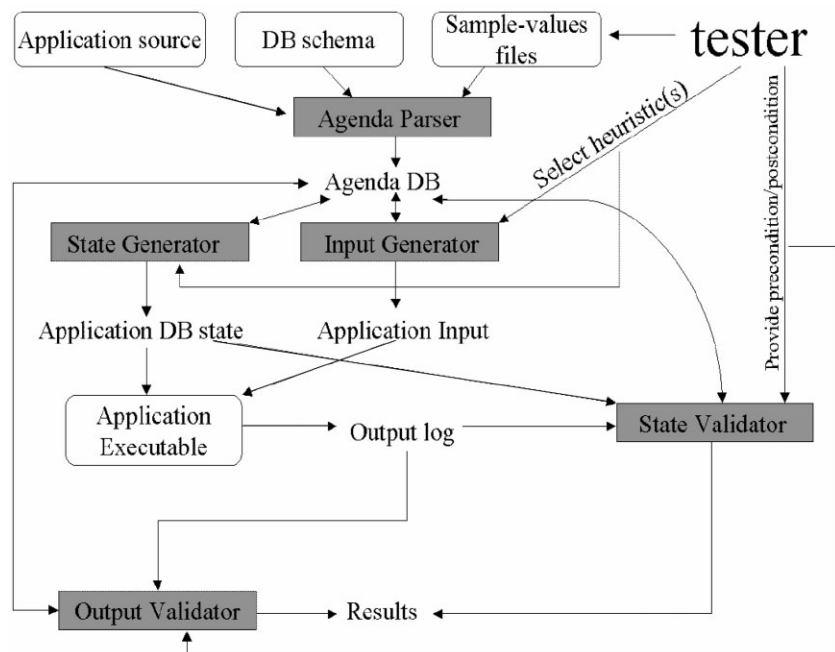


Figure 2. Architecture of the AGENDA tool set.

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. In this paper, it is assumed that the application consists of a single SQL query; work is being done to relax this assumption.

The approach is loosely based on the category-partition method [5]: the user supplies suggested values for attributes, partitioned into groups, called *data groups*[§]. These values are provided in the sample-values files. The tool then produces meaningful combinations of these values in order to fill database tables and provide input parameters to the application program. Data groups are used to distinguish values that are expected to result in different application behaviour, e.g. different categories of employees. Additional information about data groups can also be provided via annotations, as described by Chays *et al.* [2].

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 behaviour of the application on tests represented by each template. For example, the tester

[§]In the category-partition method, these data groups are called 'choices'.



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 favour ‘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 doing so is not precluded by uniqueness constraints, and heuristics to force the inclusion of values from all data groups.

Finally, AGENDA instantiates the templates with specific test values, executes the test cases and checks that the outputs and new database state are consistent with the expected behaviour indicated by the tester. In the remainder of this section, the design and implementation of AGENDA are described in more detail.

The first component (*AGENDA Parser*) extracts relevant information from the application’s database schema, the application queries, and tester-supplied sample-values files, and makes this information available to the other four components. It does this by creating an internal database, called the *AGENDA DB*, to store the extracted information. The AGENDA DB is used and/or modified by the remaining four components.

The second component (*State Generator*) uses the database schema along with information from the tester’s sample-values files indicating useful values for attributes (optionally partitioned into different groups of data), and populates the database tables with data satisfying the integrity constraints. It retrieves the information about the application’s tables, attributes, constraints, and sample data from the AGENDA DB and generates an initial database state for the application, referred 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 query in the application program. 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 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 subsections, the schema and queries in Figure 1 are used to illustrate the operations of each component. The UPDATE query involves a possible change in the application’s database state, so the components involved are: AGENDA Parser, State Generator, Input Generator, and State Validator. The SELECT query should not change the application’s database state, so the components involved are: AGENDA Parser, State Generator, Input Generator, and Output Validator.



3.2. AGENDA parsing tool

At the core of the AGENDA Parser is an SQL parser. This tool has been based on PostgreSQL, an object-relational DBMS, originally developed at UC Berkeley [6] and now commercially supported by PostgreSQL [7]. PostgreSQL supports most of SQL2 (along with additional features that are more object oriented) and provides a well documented open-source parser. The PostgreSQL parser creates an Abstract Syntax Tree that contains relevant information about the tables, attributes, and constraints. However, this information is spread out in the tree, making it inconvenient and inefficient to access during test generation. Furthermore, it is possible to use different SQL DDL syntactic constructs to express the same underlying information about a table; consequently, the location of the relevant information in the tree is dependent on the exact syntax of the schema definition. For example, the primary key constraints on tables `dept` and `emp` in Figure 1(a) are expressed using different syntactic constructs, leading to different parse sub-tree structures.

For these reasons, rather than forcing AGENDA components to work directly with the Abstract Syntax Tree, the PostgreSQL parser was modified so that while it parses the schema definition for the database underlying the application to be tested, it also collects relevant information about the tables, attributes, and constraints. An earlier version of the tool, described by Chays *et al.* [2], stored this information in a complex, dynamically expanding data structure. The current version of AGENDA Parser stores this information in a database, called the AGENDA DB, that is internal to the system. Thus, the memory issues associated with the original data structure, as discussed by Chays *et al.* [2], are avoided, and the information is more accessible to the other components of AGENDA, which are now database applications. In addition, this design choice has made it much easier to modify all the AGENDA components.

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

The PostgreSQL parser is generated by YACC from a collection of grammar rules and associated actions. The AGENDA Parser was created by modifying selected actions, and traversing selected structures, in order to insert relevant data into the AGENDA DB when it is identified during parsing.

As the AGENDA Parser parses the (application) schema, it extracts information about tables, attributes, and constraints. This version of the AGENDA Parser extracts information about uniqueness constraints and referential integrity constraints (including composite constraints involving multiple attributes), and not-NULL constraints. It also extracts limited information from semantic constraints, namely boundary values from sufficiently simple Boolean expressions. For the table `emp` in



Figure 1(a), 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.

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

Next the application query is parsed, in order to extract information about the query's parameters. This information is needed for test generation and output checking. For queries embedded in a host program, the parser must extract information about *host variables*, which are variables in the host language that are used as parameters in SQL queries. Again, the DBMS parser was modified to accomplish this. A small modification to the lexical analyzer was needed in order to allow the parser to accept queries with uninstantiated host variables. Information is extracted by traversing the query tree built by the DBMS. Clauses with complicated expressions and many input and output host variables can be handled cleanly.

The query tree is traversed to identify input and output host variables. Input host variables are found in the WHERE clause of a query and the SET clause of an UPDATE query. Output host variables are found in the INTO clause of a SELECT query. For example, as the SELECT query in Figure 1(b) is parsed, the AGENDA Parser stores in the AGENDA DB the following information: `out_name` and `out_bonus` are output host variables associated with attributes `ename` and `bonus`, respectively, belonging to the table `emp`, and `in_deptno` is an input host variable associated with attribute `deptno` of table `emp`. For the UPDATE query in Figure 1(b), the AGENDA Parser stores data indicating that `in_empno` is an input host variable associated with `emp.empno`, and `rate` is an input host variable associated with `emp.salary`. If there is an association between an input host variable and an attribute, the type of association is also stored in the AGENDA DB. A *direct* association between an input host variable and an attribute indicates to the Input Generator that when it instantiates a value for this input host variable, it can choose a sample value among those supplied by the tester for the associated attribute. 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 3.4.

Boundary values can also be easily identified in the query tree. For the `select` query in Figure 1(b), the AGENDA Parser extracts the boundary values 7000.00 and 9000.00 along with the associated attribute (`salary`) from the query tree and stores this information in the AGENDA DB. There may also be boundary values defined in the schema, as discussed above. Once extracted and stored in the AGENDA DB, these 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 above. 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.



<i>table Tables</i>		
Table	Number of attributes	Order to fill
dept	3	0
emp	5	1

<i>table Attributes</i>		
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

<i>table Boundary values</i>		
Attribute	Value	Operator
emp.salary	6000.00	\geq
emp.salary	10000.00	\leq

Figure 3. Information in the AGENDA DB after parsing the schema.

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

With reference to the application schema in Figure 1(a) and the update query in Figure 1(b), suppose it is known 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 that only test cases for host variable `:in_empno` are generated. By parsing the query, it is determined that it is associated with attribute `empno` in table `emp`. Suppose in the sample-values file for `empno`, the sample values provided for attribute `empno` are partitioned into three groups: student, faculty, and administrator, as in Figure 4. The AGENDA Parser stores information in the AGENDA DB for this schema, as shown in Figure 3.

Based on parsing the application query and sample-values files, the AGENDA Parser stores information in the AGENDA DB, indicating that the query has host variables `:rate`, indirectly associated with attribute `salary`, and `:in_empno` directly associated with `empno`; that it potentially changes attribute `salary` in table `emp`; and that there are three data groups for attribute `emp.empno`: student, faculty, administrator. It generates three templates, `template_emp_empno_student`, `template_emp_empno_faculty`, `template_emp_empno_administrator`, and stores two boundary values for attribute `salary`: 5000.00, 10000.00.



3.3. State generation tool

The State Generator populates the database. The tool has been designed to allow flexibility so that it can be used for experimentation with different test generation strategies. It retrieves relevant database schema information from the AGENDA DB, and outputs a consistent database state, in which all constraints recognized by the AGENDA Parser are obeyed. It incorporates various heuristics to aid in generating useful database states (i.e. states deemed likely to expose application faults).

For example, the tester might provide the files shown in Figure 4 for testing the application whose schema and queries are shown in Figure 1. In the file `empno`, the tester has supplied 15 possible values, partitioned into three data groups: student, faculty, and administrator.

This approach to generating values for attributes was chosen because checking test results cannot be fully automated unless a complete formal specification is available and it will often be easier for the human checking the test results to work with meaningful values, rather than with randomly generated values. Sometimes manual generation of attribute values is burdensome, so AGENDA also provides automatic derivation of sample-values files, divided into meaningful data groups, for types integer, float, and money. Suppose, as in Figure 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 $[5000.00, \dots, 6000.00)$, $[6000.00, \dots, 7000.00]$, $(7000.00, \dots, 9000.00]$, and $(9000.00, \dots, 10000.00]$ 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 [8]. Groups containing each of these values by themselves as well as separate groups containing intermediate and off-by-one values are automatically generated as shown in Figure 4. 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. Future work will examine 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. The tool 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 the 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.

Uniqueness constraints are handled as follows. The AGENDA DB table `data_group_recs` (see Appendix) has one row for each data group. The AGENDA DB table `value_recs` has one row for each value. If there is a uniqueness constraint on a single attribute, the appropriate frequency fields in the `data_group_recs` and `value_recs` tables are checked to avoid selecting the same value more than once. When a data group and value are selected for the database, the corresponding fields `choice_freq_in_db` and `val_freq_in_tc` are incremented, respectively.

Referential integrity (foreign key) constraints are handled as follows. When selecting a value for attribute `a` in table `T`, where this attribute references attribute `a'` in table `T'`, the State Generator refers



deptno:	dname:	bonus:	rate:
—choice_name: deptno	—choice_name: d1	—choice_name: bonus	—choice_name: low
10	accounting	50.00	1.01
20	—	500.00	1.05
30	—choice_name: d2	1000.00	1.07
40	research	2000.00	—
50	—	10000.00	—choice_name: high
60	—choice_name: d3		1.50
70	sales		1.75

salary:	empno:	ename:	loc:
—choice_name: exterior_OFF_point_1	—choice_name: student	—choice_name: ename	—choice_name: domestic
4999.99	111	Smith	—choice_prob: 90
—choice_name: ON_boundary_value_1	112	Jones	Brooklyn
5000.00	113	Blake	Florham Park
—choice_name: interior_OFF_point_1	114	Clark	Middletown
5000.01	115	Adams	—
—choice_name: interboundary_values_1	—	Davis	—choice_name: foreign
5168.40	—choice_name: faculty	Flanders	—choice_prob: 10
5310.53	550	Martinez	Athens
—choice_name: exterior_OFF_point_2	555	Williams	Bombay
5999.99	565	Fox	
—choice_name: ON_boundary_value_2	569	Rivera	
6000.00	570	Hernandez	
—choice_name: interior_OFF_point_2	—	Ullman	
6000.01	—choice_name: administrator	White	
—choice_name: interboundary_values_2	811	Widger	
6056.29	812		
6230.12	813		
—choice_name: exterior_OFF_point_3	814		
6999.99	815		
—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. Input files for department–employee database.

to the value record associated with the attribute record 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. New algorithms for breaking cycles are being considered. Further implementation details are provided elsewhere by Chays *et al.* [9].



3.3.1. 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 database schema and queries, such as the values 5000.00, 6000.00, and 10000.00 for the attribute `salary` in Figure 1, can be very useful since 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 select 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 database 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.

3.3.2. Example

This section presents an example illustrating a database state produced by the tool for the department-employee schema shown in Figure 1(a). Sample-values files supplied by the tester or automatically generated (for the `salary` attribute) are shown in Figure 4.

The tester is prompted to select one or more heuristics to guide state generation. 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 5. These have been obtained by executing the `insert` statements generated by the tool 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 sample-values 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 manually



<i>table dept</i>		
deptno	dname	loc
10	NULL	Brooklyn
20	accounting	NULL
30	research	Athens
40	sales	Florham Park

<i>table emp</i>				
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 5. A database state produced by the tool.

or executing test cases that select NULL values may lead the tester to question whether those attributes should have had not-NULL constraints.

3.4. Input generation tool

The Input Generator generates test cases by instantiating the application's inputs with actual values. A test case for such a query consists of instantiating all input parameters of the query. The input host variables are clearly marked with a preceding colon in the WHERE and SET clauses of each application query to be tested. For example, in the queries shown in Figure 1(b), the input host variables are `:rate` and `:in_empno`.

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



from the selected heuristics. Handling of the `boundary values` and `nulls` heuristics by the Input Generator is similar to that of the State Generator. A complete explanation is provided by Chays *et al.* [9].

If the state generation was guided by the `duplicates` heuristic, then the generated Application DB will contain some tuples that have the same value of a specific attribute. 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 on a specific attribute. The heuristic `duplicates` from the Application DB state means that the value 111, which appears more than once in the Application DB, should be chosen. This value need not appear in more than one of the test cases.

If the tester selects the `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. Implementation details, including a detailed discussion of the handling of composite constraints, are provided by Chays [10].

3.4.1. Example

The Input Generator produces test cases by instantiating values for the application query's parameters, or input host variables. The UPDATE query in Figure 1(b) 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` are partitioned into three groups: `student`, `faculty`, and `administrator`, as in Figure 4. It then finds that `rate` is not directly associated with any attribute but the tester has supplied sample values, partitioned into two groups, `low` and `high`, as in Figure 4. Since there are two input parameters, one with 3 groups and the other with 2 groups, there are a total of six test templates for test case generation: (`student`, `low`), (`student`, `high`), (`faculty`, `low`), (`faculty`, `high`), (`administrator`, `low`) and (`administrator`, `high`).

Since the tester selected `all groups` as opposed to `all templates`, the Input Generator knows that it is sufficient for each group to be represented among the test cases, rather than all combinations of all groups. This is accomplished by marking groups that are used in test cases in the AGENDA DB, just as the State Generator marks groups that are used in the DB state. The least frequently used group is given preference when instantiating its associated parameter, thus reducing the number of test cases needed to satisfy all groups. For this example, only three test cases were sufficient 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).

3.5. State validation tool

After a test case is executed, the output consists of the output returned to the user, along with the modified database state. Unless there is a formal specification of the intended behaviour 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, which are maintained by the DBMS and contain information about the modifications made to the database by the application.

3.5.1. 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 [1]. Triggers are used as the means of capturing state changes as follows.

- The schema is modified 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.
- Triggers are generated 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.
- The log tables are queried to obtain information about the changes made to the database by the application.

Triggers can be used 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 1(a), 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 only the old values are stored 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 [7].

Three rules are added to each table, corresponding to updating, inserting, or deleting. For example,

```
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, 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 for the rows in table `emp` that have been updated, inserting one row into table `emp_log` for each time table `emp` was 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.

3.5.2. Checking the state change by means of database constraints

Several issues must be considered when checking the database state.

- Checking that tables that should not have changed did not 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 [11]. Various semantic integrity constraints are supported in SQL standards. Check/assertion constraints are particularly useful when validating 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.

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.

3.5.3. Example

Consider the schema in Figure 1(a) and the UPDATE query in Figure 1(b). The AGENDA Parser extracts the information as described in Section 3.2. Assume it has been determined (e.g. by doing some data flow analysis on the source code) that the value of host variable `:rate` is determined by the application so test values are only generated for host variable `:in_empno`.

The tool asks the tester to specify the expected result for each template. In this case, only one input host variable, with three data groups, is considered; so there are three 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 three 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. Since 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 the AGENDA 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. After the tool completes the test for the test case, it removes the constraint:

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

An example involving a test case in which a value is selected by the Input Generator that is not in the Application DB, according to the heuristic values not in Application DB, is discussed by Chays *et al.* [9].



3.6. Output validation tool

The output returned to the user may contain more than one row in the application tables, so AGENDA needs 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, a trigger/rule cannot be used 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 the 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.

3.7. Performance and limitations

This subsection discusses the performance and limitations of the AGENDA prototype.

3.7.1. Overhead estimation of the rules

In order to investigate how the addition of the rules/triggers affects the system performance, the execution time during various operations was measured, with and without the logging rules. A total of 10 000 insertions, 10 000 updates, and 10 000 deletions were run, each on three tables T5, T20, and T100, where T5 has five integer attributes, T20 has 20 integer attributes and T100 has 100 integer attributes. All of the 10 000 operations have been treated as both a single transaction, and as 10 000 transactions. The time to execute these transactions is shown in Table I. The third and fourth columns show the total cost in seconds without rules and with rules, respectively. The last column shows the overhead associated with using rules, defined as the difference between execution time with rules and without rules divided by the execution time without rules, represented as a percentage.

With one exception (T5 versus T20, insertion, 10 000 transactions), the percentage overhead increased as the number of attributes increased. When the modifications were done as 10 000 separate



Table I. Time (s) for executing transactions with and without rules.

		Time without rule	Time with rule	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 / 10 000 transactions	insert	927	1214	31
	update	1277	1588	24
	delete	697	1079	55
T20 / 10 000 transactions	insert	1016	1320	30
	update	1299	1752	35
	delete	701	1098	57
T100 / 10 000 transactions	insert	1366	2058	51
	update	1645	2612	59
	delete	830	1591	92

transactions, the overhead ranged from 24% to 92% for all operations on the three tables. In general, if the operation only changed a small portion of the database (such as updating 1 row of a table with 10 000 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, the amount of overhead seems reasonable.

The 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 retaining only the initial values of logged attributes in the log tables and/or by hashing attribute values and storing those, and then comparing them to hashes of the new values.

3.7.2. Costs of running the tools

The costs of executing AGENDA's tools on five examples are shown in Table II. Each entry is an average over five executions. The setup time consists primarily of removing old information (from prior runs) from the AGENDA DB tables. The checking time includes the execution time for running the test cases and the time for validating the results. The total numbers of rows and test cases produced by the State Generator and Input Generator, respectively, are provided. All experiments were performed on the platform of DELL Dimension 811. The CPU is a Pentium 4 Processor at 1.3 GHz. Main memory



Table II. Time (ms) for running AGENDA's tools.

	cia	movie	emp	emp1	emp2
Setup time (ms)	6820	6156	7520	7099	8190
AGENDA Parser time (ms)	12 767	16 352	17 132	16 733	18 692
State Generator time (ms)	2665	9589	6324	8216	19 534
Input Generator time (ms)	2281	1948	2883	11 064	177 221
Checking time (ms)	3309	4736	5476	25 316	145 342
Number of rows generated	3	30	12	15	29
Number of test cases generated	2	1	3	42	252
State Generator time/row (ms)	888	320	527	548	674
Input Generator time/test case (ms)	1140	1948	961	263	703
Checking time/test case (ms)	1654	4736	1825	603	577

Table III. Additional information about the examples.

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

is 256 MB. The system is implemented as a Web database application. The overhead includes the additional cost of automatic execution of the AGENDA application as specified in XML configuration files. Summary information is provided in the last three rows of Table II: time per row to generate the Application DB, time per test case to generate the inputs and time per test case to check the results.

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



3.7.3. Limitations

The AGENDA Parser accepts any database schema (full SQL), since the core of the tool is the DBMS SQL parser, but the current tool set implementation does not take full advantage of certain SQL constructs which may contain useful information for the purpose of testing. For the purpose of generating test data for the database state and test cases for the application, AGENDA utilizes all uniqueness constraints, referential integrity constraints, not-NULL constraints and composite key constraints. It uses semantic constraints to a limited extent (for extracting boundary values, as discussed in Section 3.3), but not complicated semantic and domain constraints. The current tool set implementation extracts useful information from individual queries, such as the input and/or output parameters, the table and attribute with which each parameter is directly/indirectly associated, the type of query and which table is potentially changed by the query if it is an UPDATE or INSERT. Since source code analysis and a data flow graph are not yet part of the tool set, it is assumed for now that the user supplies information about data flow and which input parameters need to be instantiated for which queries. Some input parameters do not need to be instantiated because their values are determined by statements executed before the query under test. Recently, AGENDA has been extended to handle more complicated semantic constraints, in the context of testing database transaction consistency and concurrency [12,13]. Also, the AGENDA prototype system has been demonstrated [14].

4. RELATED WORK

There is little work in the software testing research literature on testing techniques targeted specifically toward database applications. Davies *et al.* [15] populate a database using a prototype that requires the user to provide validation rules to specify constraints on attributes. Tsai *et al.* [16] automatically generate test cases from relational algebra queries. There are also some practical guidelines for practitioners, as given by Bourne [17].

Several techniques and tools have been proposed for automated or partially automated test generation for imperative programs. Most of these attempt the difficult task of identifying constraints on the input that cause a particular program feature in an imperative program to be exercised and then use heuristics to try to solve the constraint [18–21].

The approach of Chan and Cheung [22] 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 the Input Generator, but is not as complete as the approach presented in this paper 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 *et al.* [23] 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.

The technique presented in this paper is more closely related to techniques used for specification-based test generation, e.g. the work of Weyuker *et al.* [24]. The partially automated category-partition technique [5], introduced by Ostrand and Balcer, is close in spirit to the test generation technique



proposed in this paper. Dalal *et al.* [25] introduced another closely related requirements-based automated test generation approach and tool, which they call model-based testing. Unlike category-partition testing or model-based testing, however, the document that drives the technique presented here 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 [26,27]. Another aspect of performance testing is addressed by Slutz [28], 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' DBMSs in order to test those systems for correctness.

Gray *et al.* [29] have considered how to populate a table for testing database system performance, but their goal is to generate a huge table filled with dummy data having certain statistical properties. In contrast, the quantity of data in each of the tables is of far less interest in this work. The primary interest is rather to assure that there is reasonably 'real looking' data for all of the tables in the database, 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 the relevant previous work, perhaps the closest is an early paper by Lyons [30]. 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 [31]. To use DB-Fill, the tester must produce a definition file describing the schema in some special purpose language. Another commercial tool, DataFactory [32], 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 with respect to an attribute. Like the approach in this paper, 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 the new 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, the new 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, the new approach allows the tester to guide the generation of both the database state and inputs for the application (by selecting heuristics), and helps the tester to determine how the database state changes when the application is executed; the related approaches do not provide support for these important aspects of testing.

5. CONCLUSIONS

In response to a lack of existing approaches specifically designed for testing database applications, a framework has been proposed, and a tool set has been designed and implemented 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 database application and database 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, which partially automates generation of a database and test cases, and assists the tester in checking the results.

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

As demonstrated, AGENDA can handle not-NULL, uniqueness, referential integrity constraints, and semantic constraints involving simple expressions. Feedback from the tester can be used to handle constraints that may not be explicitly included in the schema. Constraints that are *not* part of the schema can also be used 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.

Currently AGENDA populates the database with either attribute values supplied by the tester or boundary values extracted from the database schema and/or database application, depending on the tester's choice. The interaction between the values used to populate the database and the values the tester enters as inputs to the application program was investigated. Although 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 database state, it is recommended that they also be used in the generation of the inputs. For example, if the `nulls` heuristic is chosen to guide the State Generator, then nulls will appear in the database 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. To address this important issue, ways of validating the output of test cases as they are executed were developed. 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. The State Validator and Output Validator were designed and implemented. 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. AGENDA's performance on several small case studies was discussed. More extensive empirical studies are in progress.

The version of AGENDA described in this paper is targeted toward relatively simple applications consisting of single queries. This provided understanding of the fundamental issues in testing database



applications. Ongoing work is extending AGENDA to handle transactions consisting of multiple queries [12], concurrent executions of transactions by different clients [13], and applications embedded in a high-level language.

ACKNOWLEDGEMENTS

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

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.

APPENDIX. AGENDA DB SCHEMA

Following are some relevant AGENDA DB tables. An example illustrating some of the values stored in these tables is provided in Section 3.2.

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

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

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

```
create table attribute_recs (table_name varchar(25), attr_name varchar(25),
data_type int, size int default 0,
is_primary bool not null default 'F', is_unique bool not null default 'F',
is_not_null bool not null default 'F',
is_check bool not null default 'F', is_default bool not null default 'F',
is_composite bool not null default 'F', is_boundary bool not null default 'F',
foreign_table varchar(25), foreign_attr varchar(25),
num_groups int not null default 0,
prob_or_freq varchar(4) not null default 'PROB',
primary key(table_name, attr_name),
foreign key(table_name) references table_recs,
check (foreign_table!='' OR foreign_attr!=''),
check (foreign_attr!='' OR foreign_table!=''),
check (prob_or_freq in ('PROB','FREQ')));
```

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



```

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

```

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

```

create table value_recs (table_name varchar(25), attr_name varchar(25),
group_name varchar(25), value varchar(25),
val_freq_in_db int default 0 not null,
val_freq_in_tc int default 0 not null,
fk_val_freq_in_db int default 0 not null,
index int not null default 0, random_index int not null default 0,
check(val_freq_in_db>=0), check(val_freq_in_tc>=0),
check(index>=0), check(random_index>=0));

```

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

```

create table composite_choice_recs (composite_index int default -1,
composite_type varchar(2) not null default 'CP', table_name varchar(25),
num_attributes int default 2,
attr varchar(25), data_group varchar(25),
num_values int default 0, choice_percent float4 default 100.0,
comp_choice_want_in_db int not null default 0,
comp_choice_want_in_tc int not null default 0,
comp_choice_freq_in_db int not null default 0,
comp_choice_freq_in_tc int not null default 0,
check(composite_type in ('CP', 'CU', 'CF')),
primary key(composite_index, composite_type, table_name, attr),
foreign key(table_name) references table_recs,
foreign key(table_name, attr) references attribute_recs,
check(choice_percent>=0 and choice_percent<=100),
check(comp_choice_freq_in_db>=0),
check(comp_choice_freq_in_tc>=0));

```

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



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

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

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

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

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

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

```
create table template_info (xact_ID int, name varchar(500),
expected_num_row int, num_row int, test_constraint varchar(200),
tableNum int, tableNames varchar(100), result int, test int,
primary key(xact_ID, name));
```

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

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



REFERENCES

1. Elmasri R, Navathe SB. *Fundamentals of Database Systems* (3rd edn). Addison-Wesley: Boston, MA, 2000.
2. Chays D, Dan S, Frankl PG, Vokolos FI, Weyuker EJ. A framework for testing database applications. *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, August 2000. ACM Press: Boston, MA, 2000; 147–157.
3. Date CJ, Darwen H. *A Guide to the SQL Standard*. Addison-Wesley: New York, 1997.
4. Garcia-Molina H, Ullman JD, Widom J. *Database System Implementation*. Prentice-Hall: Englewood Cliffs, NJ, 2000.
5. Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 1988; **31**(6):676–686.
6. Stonebraker MR, Rowe LA. The design of postgres. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, June 1986. ACM Press: New York, 1986; 340–355.
7. PostgreSQL Global Development Group. PostgreSQL. <http://www.postgresql.org> [1999].
8. White LJ, Cohen EI. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering* 1980; **6**(3):247–257.
9. Chays D, Deng Y, Frankl PG, Dan S, Vokolos FI, Weyuker EJ. AGENDA: A test generator for database applications. *Technical Report TR-CIS-2002-04*, Department of Computer Science, Polytechnic University, Brooklyn, New York, 2002.
10. Chays D. Test data generation for relational database applications. *PhD Thesis*, Department of Computer Science, Polytechnic University, Brooklyn, New York, 2003.
11. Lewis PM, Bernstein A, Kifer M. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley: Boston, MA, 2002.
12. Deng Y, Frankl PG, Chays D. Testing database transaction consistency. *Technical Report*, Department of Computer Science, Polytechnic University, Brooklyn, New York, 2003.
13. Deng Y, Frankl PG, Chen Z. Testing database transaction concurrency. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society Press: Los Alamitos, CA, 2003; 184–193.
14. Chays D, Deng Y. Demonstration of AGENDA tool set for testing relational database applications. *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos, CA, 2003; 802–803.
15. Davies RA, Beynon RJA, Jones BF. Automating the testing of databases. *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000. IEEE Computer Society, 2000.
16. Tsai WT, Volovik D, Keefe TF. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering* 1990; **16**(3):316–324.
17. Bourne KC. *Testing Client/Server Systems*. McGraw-Hill: New York, 1997.
18. DeMillo RA, Offutt AJ. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(2):109–127.
19. Korel B. Automated software test generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–879.
20. Gotlieb A, Botella B, Rueher M. Automatic test data generation using constraint solving techniques. *Proceedings of the 1998 International Symposium on Software Testing and Analysis*, March 1998. ACM Press: New York, 1998; 53–62.
21. Gupta N, Mathur A, Soffa ML. Automated test data generation using an iterative relaxation method. *Proceedings of the Conference on Foundations of Software Engineering*, November 1998. ACM Press: New York, 1998; 231–244.
22. Chan MY, Cheung SC. Testing database applications with SQL semantics. *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, March 1999. Springer: New York, 1999; 363–374.
23. Zhang J, Xu C, Cheung SC. Automatic generation of database instances for white-box testing. *Proceedings of the 25th Annual International Computer Software and Applications Conference*, October 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 161–165.
24. Weyuker EJ, Goradia T, Singh A. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering* 1994; **20**(5):353–363.
25. Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM, Patton GC. Model-based testing in practice. *Proceedings of the 21st International Conference on Software Engineering*. ACM Press: New York, 1999; 285–294.
26. Transaction Processing Performance Council. TPC-Benchmark C. <http://www.tpc.org> [1998].
27. Carey MJ, DeWitt DJ, Naughton JF. The 007 benchmark. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993. ACM Press: New York, 1993; 12–21.
28. Slutz D. Massive stochastic testing of SQL. *Proceedings of the Conference on Very Large Databases*, August 1998. Morgan Kaufmann: San Francisco, 1998; 618–622.
29. Gray J, Sundaresan P, Englert S, Baclawski K, Weinberger PJ. Quickly generating billion-record synthetic databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 1994; **23**(2):243–252.
30. Lyons NR. An automatic data generating system for data base simulation and testing. *Database* 1977; **8**(4):10–13.
31. DB-Fill. <http://www.bossi.com/dbfill> [June 2002].
32. DataFactory. <http://www.quest.com/datafactory> [June 2002].