Query-aware Test Generation Using a Relational Constraint Solver

Shadi Abdul Khalek Bassem Elkarablieh Yai O. Laleye Sarfraz Khurshid
The University of Texas at Austin
{sabdulkhalek, elkarabl, lalaye, khurshid}@ece.utexas.edu

Abstract

We present a novel approach for black-box testing of database management systems (DBMS) using the Alloy tool-set. Given a database schema and an SQL query as inputs, our approach first formulates Alloy models for both inputs, and then using the Alloy Analyzer, it generates (1) input data to populate test databases, and (2) the expected result of executing the given query on the generated data. The Alloy Analyzer results form a complete test suite (input/oracle) for verifying the execution result of a DBMS query processor.

By incorporating both the schema and the query during the analysis, our approach performs query-aware data generation where executing the query on the generated data produces meaningful non-empty results.

We developed a prototype tool, ADUSA, and used it to evaluate our approach. Experimental results show the ability of our approach to detect bugs in both open-source as well as commercial database management systems.

I. Introduction

The use of database management systems (DBMSs) to manage and manipulate data is ubiquitous. DBMSs are steadily growing in complexity and size. At the same time reliability is becoming a more vital concern; the cost of user data loss or wrong query processing is prohibitively expensive [10]. DBMS testing, in general, is a labor intensive, time consuming process, often performed manually. For example, to test the correctness of a query execution, the tester is required to populate the database with interesting values that enable bug discovery, and manually check the execution result of the query based on the input data. Automating DBMS testing not only reduces development costs, but also increases the reliability in the developed systems.

Automatic DBMS testing includes generating (1) test queries for a given database schema, (2) a set of test databases, and (3) oracles to verify the result of executing the queries on the input databases using the DBMS.

Several approaches exist for automatic query generation. For example, RAGS [21] and QGen [20] stochastically combine SQL statements to generate valid queries. These tools enable generating thousands of SQL queries in a few seconds. A more recent approach [5] targets generating queries with cardinality constraints. Integrating query generators with data generators, however, is still either specialized [20], or sometimes not possible [5].

Several academic and commercial tools target the problem of test database generation [4], [7], [11], [22]. Given a database schema, constraints on the table sizes, and generation rules such as ranges for the values or statistical properties, these tools generate databases that satisfy the given schema and constraints. Such generation methods, however, are *query unaware*, i.e., they don't take the queries (which usually relate to the components under test) into consideration while generating the data, and thus, executing these queries might not return meaningful results. Recent approaches introduced *query-aware* database generation [2], [3]. These approaches use the information from the queries to constrain the data generator to generate databases that provide interesting results upon query executions. Query-aware generation is gaining popularity in both DBMS and database application testing [9], [25].

Database generation approaches, however, do not provide a mechanism to verify the result of executing test queries on the generated databases. This process is currently performed either manually, or by differential testing, i.e., comparing the execution results of a previous version of the DBMS with the current version to verify that any changes did not break a working component.

In this paper, we present a novel approach for effective black-box DBMS testing. Our approach uses model-based testing to perform (1) *query-aware* database generation to construct a useful test input suite that covers the various scenarios for query execution and (2) test oracle generation to verify query execution results on the generated databases. As an enabling technology, we use Alloy [12], [14], an analyzable relational language, and the Alloy Analyzer. Alloy's relational basis provides a natural fit for modeling relational databases and query operations.

978-1-4244-2188-6/08/$25.00 ©2008 IEEE

Given an input database schema and an input SQL query (manually written by the user, or automatically generated by an existing tool), our approach formulates Alloy specifications to model both of the inputs, and then uses the Alloy Analyzer to generate all bounded non-isomorphic databases that satisfy the given schema, as well as the expected result of executing the given query on each of the generated databases. Each input/oracle pair is then used to test the query execution on the DBMS.

We developed a prototype tool, ADUSA, and used it to evaluate our approach on testing the query execution correctness of three DBMSs including two open source systems, HSQLDB [23] and MySQL, and one commercial DBMS, Oracle11g. Experimental results show that ADUSA was able to detect (1) a bug in the production release of Oracle11g Enterprise Edition*, (2) bugs that are previously reported in the MySql bug repository, and (3) bugs that we injected in HSQLDB.

ADUSA automates test input generation and test oracle generation. This enables building a unified framework for both DBMS testing (as presented in this paper) as well as database application testing. For example, using ADUSA in conjunction with an SQL query generator enables full automation of the testing process. As another example, ADUSA can be used with concolic analysis [9] of database applications. Before a program executes a query on a DBMS, ADUSA can populate the databases with various tables that enable discovering different program paths.

We make the following contributions:

- **Query-aware database generation:** We present a framework for query-aware test database generation using Alloy. Our framework incorporates the query information to generate meaningful test cases for evaluating the correctness of a query execution.
- **Test oracle generation:** We present a framework for test oracle generation which enables verifying the result of query execution on the DBMS under test.
- **Alloy for DBMS testing:** We introduce the use of Alloy specifications to model database schemas and queries. We also present the use of SAT technology for database testing.
- **Evaluation:** We evaluated our approach on finding bugs in both open-source as well as commercial database management systems.

II. Overview: Alloy for DBMS testing

In this section, we present the key insight for using Alloy in DBMS testing. We first illustrate our testing approach with an example and then discuss the advantages of using Alloy for both test input and oracle generation.

*We are checking with Oracle if the bug is known to them or not.

A. Example

We describe how to generate an Alloy specification to model a database schema and a test query, and use the Alloy specification to test a DBMS. We next describe the subset of Alloy that we use throughout this paper. More details about Alloy can be found in [13], [24].

Alloy is a strongly typed specification language. It assumes a universe of atoms (or elements) partitioned into subsets, where every type is associated with a subset. An Alloy specification consists of a sequence of paragraphs where a paragraph enables defining new types, introducing relations between types, and adding constraints on these types and relations. Being an analyzable relational language, Alloy semantics are closely related to those of relational databases. This enables systematic modeling of relational databases, and automated analysis of relational operations on databases.

Alloy can be used to model a relational database schema. To illustrate, consider the following SQL statement for describing a database schema:

```
CREATE TABLE student (
  id int,
  name varchar(50),
  PRIMARY KEY (id)
);
```

The above SQL statement creates the student relation with two attributes, `id` of type `int`, and `name` of type `varchar`. The statement also specifies the PRIMARY KEY constraint on the `id` attribute of the relation. The PRIMARY KEY constraint restricts the number of tuples in the `student` relation with the same `id` attribute to one.

ADUSA generates an Alloy specification that represents the student schema. The following Alloy specification models the above schema representation:

```
sig varchar{}

one sig student {
    rows: Int -> varchar
} {
    all x: rows.varchar | one x.rows
}
```

Alloy provides signature paragraphs (`sig`) to declare new types. For example, the `varchar` *signature* introduces the `varchar` type which models the elements that represent the `name` attribute of the `student` table. Note that introducing the `varchar` type is required since Alloy only has an internal representation for integers.

A `sig` paragraph also enables declaring relations between types as well as constraints on these relations. For example, the `student` *signature* declares the `rows` relation from `Int` to `varchar` to model the rows of the `student` table. The domain of the `rows` relation models the `id` attribute of the `student` table, and the image of the `rows` relation models the `name` attribute. The primary key constraint on the `student` table is modeled using the Alloy statement `all x: rows.varchar | one x.rows`.

The `all` quantifier stands for universal quantification. The dot operation `'.'` represents a relational join, e.g., `rows.varchar` returns the domain of the `rows` relation and the `x.rows` returns the image of an element `x` in the `rows` relation. The Alloy constraint holds if and only if each element in the domain of `rows` has exactly one image, thus, modeling the `PRIMARY KEY` constraint.

Alloy specifications can also be used to model operations on relational databases. For example, consider the following SQL query:

```
SELECT DISTINCT id
FROM student
WHERE (id=1 OR (id>=3 AND id<=5));
```

The above query is a selection SQL statement that queries the `student` relation, and returns all the `id` elements that are either equal to 1, or between 3 and 5. The following Alloy paragraphs model the SQL query.

```
fun query () : Int {
    {select[where[from[student.rows]]]}
}

fun select (rows: Int -> varchar) : Int {
    {rows.varchar}
}

fun where(rows: Int -> varchar) : Int -> varchar {
    {x1: rows.varchar, x2: x1.rows | condition[x1]}
}

pred condition(x1: Int) {
    {eq[x1, 1] or (gte[x1, 3] and lte[x1, 5] )}
}

fun from(rows: Int -> varchar) : Int -> varchar {
    {rows}
}
```

Constraints in Alloy are modeled using *functions* (`fun`), which represent named expressions, and *predicates* (`pred`), which represent named formulas. Alloy provides two different paragraphs to represent functions and predicates. A `fun` paragraph takes a relation as input and returns a relation of similar or different arity. A `pred` paragraph takes a relation as input and evaluates to a boolean value. Bracket notation `[...]` is used for function invocations.

The above selection SQL query is modeled in Alloy as a predicate that models the condition of the `WHERE` clause as well as three functions modeling the `SELECT`, `FROM`, and the `WHERE` clauses. The predicate `condition` takes an element of type integer and checks whether the element is equal to 1, or between 3 and 5 inclusive. The `where` function takes a relation `row: Int -> varchar`, iterates over its tuples, and returns a subset of the relation that contains the tuples whose domain values satisfy the `condition` predicate. Since the target of the SQL query is a single table, the `from` paragraph takes the relation `rows: Int -> varchar` and returns it without performing any operations. Models for more complex queries that includes multiple tables will have more complex `from` functions. Finally the `select` function takes a relation `row: Int -> varchar` and returns the domain of the relation.

The function named `query`, in the above Alloy example, models the execution of the SQL query on the database. The function first invokes the `from` function on the `student` table. Then, it calls the `where` function on the results to obtain the tuples that satisfy the condition. Finally, it calls the `select` function that returns the `id` of the returned tuples.

After generating the Alloy specification for the schema and the query, ADUSA uses the Alloy Analyzer to find database instances that satisfy the specification, i.e., provide valuations for the types and relations that satisfy all the constraints. The Alloy Analyzer finds all the instances within a given scope, i.e., a bound on the number of data elements of each type to be considered while generation. Alloy provides the `run` command to specify the scope for the Alloy Analyzer. For example, using the command `run query for exactly 2 varchar` instructs the Alloy Analyzer to enumerate all the tables using two `varchar` elements. Below is an example of an instance generated by the Alloy Analyzer:

```
varchar: {varchar$0, varchar$1}
student:{student$0}
rows: {(1, varchar$1), (2, varchar$0), (4, varchar$1)}
$query: {1, 4}
```

In the above instance, the `varchar` set has two elements labeled as `varchar` followed by the element number. The rows relation consists of three tuples whereby two tuples satisfy the condition of the `condition` predicate. The `query` set represents the result of executing the query on the `rows` relation. The `query` set holds the `id` attribute of the tuples that satisfy the `condition` predicate. For this example, the result is the set of integers `{1, 4}`.

After generating the Alloy Instance, ADUSA translates the instance into `INSERT` SQL queries that are used to populate an empty database. For example, for the above Alloy instance, ADUSA identifies the `rows` relation and generates the following `SQL` statements:

```
INSERT INTO student VALUES (1, varchar$1)
INSERT INTO student VALUES (2, varchar$0)
INSERT INTO student VALUES (4, varchar$1)
```

Once the database is populated with data, the given SQL selection query is executed on the DBMS and the result is verified with the one found in the Alloy instance.

This process is repeated for each generated instance. For the example described in this section, we ran the instances on MySql5.0 DBMS, and as expected for the simple query no violations were found in a total of 36 instances.

B. Query-aware generation

By incorporating information from the SQL query and the schema, our approach performs effective query-aware generation where the generated test cases are guaranteed to produce meaningful results upon query execution as opposed to query unaware generators where the execution results are highly likely to be empty.

ID	NAME
13685	1uH
15209	17Sx
16535	8UJI7MA
29822	ZF5K5b63MJ
...	...

Table *student*

Fig. 1. A snapshot database table populated using random data generation.

To illustrate the effectiveness of the query-aware generator in ADUSA, we used it to generate a set of databases for testing the following query:

```
SELECT *
FROM student
WHERE id < 3 and name = 'John'
```

We compare the ADUSA results with those generated by a commercial database generation tool [6] that populates the database tables with randomly generated data without considering the target query. Figure 1 shows a snapshot of the generated data for a `student` table with an `id` attribute of type integer and a `name` attribute of type `varchar`. The generated random integers for the `id` variable are all large numbers, and the string values are all random strings. Executing the query on the generated data gives an empty set.

Figure 2 shows an example of 3 instances generated by ADUSA. Unlike the random integer values displayed in Figure 1, the integer values generated by ADUSA are closely related to the predicate (`id < 3`) described in the query. Based on the given predicate, ADUSA partitions the integer space into two regions: integers with values less than three and integers with values greater than or equal to three. ADUSA then generates tuples with integer values from each of the regions, as well as tuples with integer values from both regions. The same approach is performed for the string types. ADUSA generates strings that are either equal to "John" or not. The generated instances are then constructed using values from these spaces. For example, the first instance from Figure 2 only contains integers with values less than three, and strings with values equal or unequal to "John"; the second instance only contains integers with values greater than or equal to three; and the third instance contains both.

Executing the query on the generated input results in one tuple for the first instance, no tuples for the second, and two tuples for the third instance. The use of the Alloy Analyzer, which uses off-the-shelf SAT technology, enables enumerating different instances that cover the spaces for the data types involved in the query constraints.

C. Test oracle generation

Almost all academic and commercial test data generators don't provide a mechanism for verifying the result

```
Instance 1 - rows: {(0, varchar), (1, John)}
Instance 2 - rows: {(4, John), (6, varchar), (3, varchar)}
Instance 3 - rows: {(0, John), (3, varchar), (2, John)}
```

Fig. 2. Alloy instances for a database table generated using ADUSA.

of a query execution on the generated data. An obvious approach to do this is by using a trusted (golden) DBMS as an oracle. This process includes running the query on both the test and the golden DBMSs and matching the results to assure correctness. Using this approach requires the existence of a more advanced DBMS that supports the features under test as well as the correct implementation of those features.

A key advantage of using Alloy as a modeling language is the ability to generate test oracles for free. While performing data generation, the Alloy Analyzer automatically labels a subset of the generated data as the query result. Such feature provides an automatic and efficient way to verify the execution result of the DBMS under test. A more subtle advantage is that we can even specify constraints on query results, and generate input databases that satisfy these constraints, e.g., cardinality constraints on the results' size.

III. The ADUSA Framework

ADUSA is a framework for automated testing of query processors. It builds on Alloy and the Alloy Analyzer, and aims at checking the result of query execution, without having to model the query processor implementation. By using Alloy, ADUSA models the semantics of relational databases without considering specific DBMS implementation details. By using the Alloy Analyzer, ADUSA enables reasoning about the model, and generating inputs as well as oracles to test the correctness of a query execution.

A. Framework Outline

Figure 3 shows the ADUSA framework. Boxes represent the processing modules of the framework; ovals represent the inputs and the outputs of these modules. ADUSA is comprised of four key modules:

1) The `SQL2Alloy` module which generates an Alloy specification by translating a given set of SQL statements representing a database schema and a test query.
2) The Alloy Analyzer module which generates instances that satisfy the translated Alloy specification.
3) The `Alloy2SQL` module which translates the Alloy instances into SQL statements that are used to populate the database on the DBMS under test.
4) The `AdusaTest` module, the outer most box, which integrates all the modules, compares the DBMS with the Alloy query results, and reports the test status.

The rest of this section describes each of the modules.

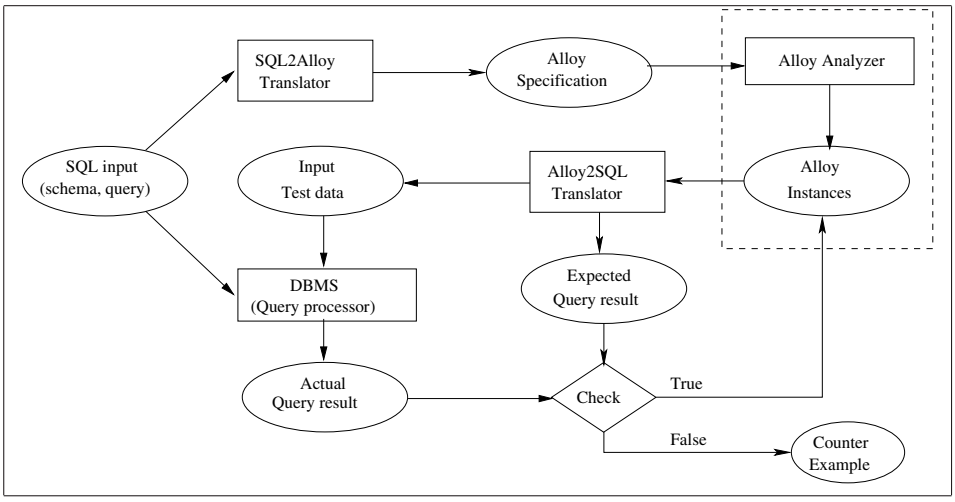

Fig. 3. ADUSA framework for DBMS testing.

B. SQL to Alloy Translation (SQL2Alloy)

ADUSA considers a subset of SQL with a simplified syntax (Figure 4). It categorizes SQL statements into two groups: *creation* statements that are used to represent the database schema, and *selection* statements that are used to retrieve data from the database without modifying its state. We next discuss how ADUSA translates these statements into an Alloy specification.

1) Creation statements: A creation statement starts with the `CREATE TABLE` keyword (Figure 4). It specifies the table name and a set of attributes representing the table columns, each associated with a data type. In addition to attributes, a creation statement defines constraints on the elements of the table. ADUSA handles two of these constraints, the `PRIMARY KEY` and `FOREIGN KEY` constraints.

ADUSA maps creation statements into a set of Alloy paragraphs as follows.

For each attribute datatype, a simple `sig` is created (analogous to the `varchar` signature in Section II-A). A `sig` declaration is added to model the table. The rows in the table are modeled using the n-ary relation `rows` among the datatypes. Primary key and foreign key constraints map into Alloy constraints added to the end of the `sig` declaration. To illustrate, consider the following SQL statements that declare two tables `tname1`, and `tname2`.

```
CREATE TABLE  tname1 (
    a1 t1,
    a2 t2,
    ...
    an tn,
    PRIMARY KEY (a1)
    FOREIGN KEY (an)
    REFERENCES tname2 (bn)
)
```

```
CREATE TABLE  tname2 (
    b1 t1,
    b2 t2,
    ...
    bn tn,
    PRIMARY KEY (bn)
)
```

Each table has `n` attributes with different datatypes. Table `tname2` has attribute `bn` as a primary key. Table `tname1` has the attribute `a1` as a primary key, and `an` as a foreign key that refers to the primary key of table `tname2`.

The above tables are translated into the following Alloy specification:

```
sig t1{}
sig t2{}
...
sig tn{}
one sig tname1 {
    rows: t1 -> t2 -> ... -> tn,
} {
    // PRIMARY KEY constraint
    all x: rows.tn.tn-1. (...).t2 | one x.rows
    // FOREIGN KEY constraint
    tn-1.tn-2.(...).t1.rows in
          tn-1.tn-2.(...).t1.(tname2.rows)
}
one sig tname2 {
    rows: t1 -> t2 -> ... -> tn,
} {
    // PRIMARY KEY constraint
    all x: tn-1.tn-2.(...).t1.rows | one rows.x
}
```

Each data type is translated into a separate `sig` declaration, and each table is translated into a `sig` declaration with a relation representing the attributes. The translation of the primary and foreign key constraints is performed through a cascaded use of the `"."` operator. For example, the expression `rows.tn.(...).t2`, returns the first domain of the rows relation corresponding to attribute `a1`; the expression `t1.t2.(...).tn-1.rows` returns the last domain of the `rows` relation corresponding to the attribute `an`; and so on. The primary key constraint of `sig tname1` specifies that each element corresponding to the `a1` attribute maps to one tuple in the table. Similarly, the primary key constraint of `sig tname2` specifies that each element corresponding to the `bn` attribute maps to one tuple in the table. The foreign key constraint of `sig tname1` specifies that the elements corresponding to `an` of table `tname1` must be a subset of those corresponding to `bn` in table `tname2`.

By translating all the creation statements, ADUSA generates an Alloy specification that models the schema.

```
CREATE TABLE  tname (
  a1 t1,
  a2 t2,
  ...
  an tn,
  PRIMARY KEY (a1[,a2, ... an])
  FOREIGN KEY (a1[,a2, ... an])
  REFERENCES tname (a1[,a2, ... an])
)
```

```
SELECT a1, a2, ... an
FROM ts
WHERE cond
```

Fig. 4. Simplified SQL syntax.

2) Selection statements: A selection SQL statement starts with the `SELECT` keyword (Figure 4), and when executed, queries the database and returns a set of tuples that satisfy a given constraint. The `FROM` clause of a selection statement specifies the target relation of the query. This relation can represent a simple table, similar to the `student` table in Section II-A, or the result of `join` operations on multiple tables. ADUSA handles both simple tables as well as the following `join` operations: `CROSS JOIN`, and `NATURAL JOIN`. The `WHERE` clause specifies the constraints that the tuples of the query relation must satisfy. Figure 5 displays the grammar of the SQL constraints that are handled by ADUSA. Briefly, ADUSA can handle relational operations on integers, equality operations on strings, as well as logical operations.

A selection SQL statement is modeled in Alloy as a set of `fun` paragraphs representing each of the clauses and a `pred` paragraph that models the constraints specified in the `where` clause. We have illustrated the modeling process with an example in Section II-A.

The Alloy language includes a collection of set operators such as Cartesian product `'->'`, relational join `'.'`, union `'+'`, intersection `'&'` and difference `'-'` as well as arithmetic and relational operations on integers which enable straightforward mapping of the SQL constraints and clauses into the Alloy functions. For example, for translating SQL queries which operate on multiple tables (or relations), ADUSA generates `fun` paragraphs that compute the Cartesian product of these relations and translates the constraints into Alloy operations on the Cartesian product. Alloy also provides the cardinality operator `'#'` which returns the number of tuples in a relation. This operation enables easy modeling of the `COUNT` SQL aggregate function, as well as specifying constraints on the cardinality of the tables to generate.

3) Modeling constants: Constants in SQL query require special modeling in Alloy since Alloy, by default, only supports primitive integers. For example, in the query described in Section II-A, the predicate invocation `eq[x1, 1]` was used to check the equality between an integer atom with an integer constant. To translate integer constants and operations on integers, ADUSA simply maps the SQL operations on integers into invocations of the Alloy functions that manipulate Alloy integers.

Strings and other SQL types are handled differently. Since Alloy does not have a built in notation for representing SQL types, ADUSA needs to automatically generate Alloy paragraphs to model these types. The current implementation handles equality operations on these types. To model an SQL type, say `varchar`, ADUSA simply adds an empty signature declaration of the type into the specification, e.g., `sig varchar`. This declaration introduces the new type as a set of uninterpreted atoms. To model constants, ADUSA adds new signatures associated with each constant defined in the query and constrains these signatures to be singletons and to be a subset of their type signature.

```
cond ::= term | cond OR term
term ::= factor | term AND factor
factor ::= NOT cond | integerterm | stringterm
integerterm ::= value op value
stringterm ::= id eq id | id eq string
value ::= num | id
op ::= eq|<|>|<=|>=
eq ::= !=|=
```

Fig. 5. Simple grammar for *cond*.

For example, consider the query described in Section II-B. To model the string constant `'John'`, ADUSA introduces the following signature `one sig John in varchar` which defines `John` as a singleton set that is a subset of the `varchar` types. Equality operations on the new introduced types are performed using the built in Alloy `'='` and `'!='` operators.

By translating the SQL creation and selection SQL statements, ADUSA generates an Alloy specification that is then analyzed using the Alloy Analyzer module to generate test instances that satisfy the specification.

C. Alloy to SQL Translation (Alloy2SQL)

ADUSA translates the Alloy instances generated by the Alloy Analyzer into SQL statements that enable populating a database with concrete values and proceedind with the testing process on the DBMS.

Recall that an Alloy instance is a set of valuations assigned by the Alloy Analyzer to the signatures and relations declared in the specification. To translate an instance, ADUSA first identifies the signatures that are associated with the database tables along with their corresponding `rows` relations. For each tuple in the `rows` relation, ADUSA generates an insertion `SQL` statement that inserts the values in the tuples into the corresponding table. For instance, in the example from Section II-A, the `(1, varchar$1)` of the `rows` relation of the `student` signature is translated into the following statement:

```
INSERT INTO student VALUES(1, varchar$1)
```

After populating the database, the testing process in ADUSA proceeds by executing the given query on the populated database and matching the result with expected output from the Alloy instance. ADUSA testing terminates when all the Alloy instances are considered or when a certain number of instances are generated.

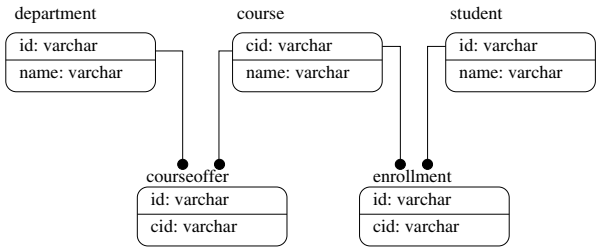

Fig. 6. Schema for a student database.

IV. Case Studies

We have used ADUSA to test a variety of database management systems including (1) Oracle11g [19], a commercially used DBMS which is widely used among business organizations, (2) MySQL [1], an open source DBMS used among web developers for its multi platform support, and (3) HSQLDB [23], a popular open source project with over 380,000 downloads from SourceForge.net. We present several case studies to illustrate the advantages of using ADUSA and we discuss some of the bugs that it detected.

A. Experiments with Oracle11g

The initial motivation behind using Oracle11g was to verify the correctness of the testing modules in ADUSA. When running ADUSA on Oracle11g (V11.1.0.6.0), we were able to detect a bug while executing a test query that involves natural join operations. We next outline the process of using ADUSA with Oracle11g. We first describe a database schema and a set of test queries that we used in the experiment. We then discuss the testing results and present the bug that the ADUSA analysis detected.

1) Database schema: Figure 6 shows the design of a student-enrollment database. The database contains five tables. The `student` table holds information about students and has 2 attributes: `id` which is the primary key and `name`. The `course` and the `department` tables are similar to the `student` table with the same attributes and with an `id` as the primary key. The `enrollment` and `courseoffer` tables record the student enrollment in courses and the departments' course offerings. The `courseoffer` table models a one-to-many relation between the `department` and the `course` tables, and the `enrollment` table models a many-to-many relationship between the `student` and the `course` tables. All the attributes in this database are of type `varchar`.

The schema in Figure 6 covers a range of the constraints that ADUSA can handle when generating an Alloy specification to model the database schema. For example, a single attribute primary key constraint is defined on the `student`, `course`, and `department` tables; a multi attribute primary key constraint is defined on the `enrollment` table; and a foreign key constraint is defined on the `student` table that refers to the `enrollment` table.

```
Q1: SELECT id
    FROM student
    WHERE name = 'John'

Q2: SELECT s.id, s.name
    FROM student s, enrollment e
    WHERE s.id = e.id

Q3: SELECT name
    FROM student NATURAL JOIN department

Q4: SELECT COUNT(*)
    FROM student NATURAL JOIN course NATURAL JOIN department
```

Fig. 7. Test queries used to evaluate ADUSA on Oracle11g.

2) Test queries: We selected a set of subject queries which vary in their syntax and execution complexity. Figure 7 displays the subject queries. The first query (`Q1`) operates on a single table and includes a comparison operation on the constant string `'John'`. The second query (`Q2`) operates on the cross product of multiple tables and includes a predicate that compares elements from different tables. The third query (`Q3`) operates on the natural joins of two tables with an empty predicate. The last query (`Q4`) operates on a cascaded natural join on three tables and includes the aggregate function `COUNT`.

The queries in Figure 7 cover a range of the SQL constructs that ADUSA can handle (Figure 5) when generating an Alloy specification to model the database query. For example, query `Q1` includes operations on constant strings; query `Q2` includes operations on the Cartesian product of multiple relations; query `Q3` includes natural join operations; and query `Q4` includes an aggregate function.

3) Experimental results: We ran ADUSA on Oracle11g using each of the described test queries. For each query, we set a scope, i.e., a bound on the number of `varchar` elements to use for generating the database instances. We ran the tests for scopes of 2, 3, and 4 elements. For the schema described in this section, the scope reflects the maximum number of records in a table.

We used one of three conditions to terminate ADUSA's execution: (1) when all the instances for the given scope have been verified, (2) upon verifying 1,000 instances, or (3) upon a test failure, i.e., an inconsistency between the Alloy and the DBMS results. Upon a test failure, we report a counter example that describes the failure scenario, i.e., the data required to reproduce the failure and the expected and actual query execution result.

Table I shows the testing results. Column 2 displays the scope used in each execution; column 3 shows the number of generated instances; column 4 shows the test result; and column 5 shows the running time in milliseconds.

The running time results illustrate the efficiency of ADUSA in verifying a query execution. For query `Q1` ADUSA generated 75 test databases and verified the execution result in less than 7 seconds, for query `Q2` ADUSA

Query	Scope	# of instances	Inconsistency ?	Time(ms)
Q1	2	4	No	358
	3	15	No	1,193
	4	75	No	6,020
Q2	2	182	No	56,108
	3	$\geq$ 1,000	No	59,592
	4	$\geq$ 1,000	No	78,577
Q3	2	83	No	6,887
	3	$\geq$ 1,000	No	33,160
	4	$\geq$ 1,000	No	39,707
Q4	2	200	Yes	10,522
	3	3	Yes	229
	4	1	Yes	$\leq$ 1

TABLE I. Results for executing ADUSA on Oracle11g using 4 test queries.

generated 1,000 test databases and verified the execution result in 78 seconds, and for query `Q3` ADUSA generated 1,000 test databases and verified the execution result in less than 40 seconds.

When running the experiment on query `Q4` with scope 2, ADUSA was able to detect an inconsistency in the query execution results and reported a counter example after generating 200 test databases, i.e., instances where the bug didn't surface. We increased the scope to 3 and re-ran the experiment. ADUSA generated a counter example after 3 valid instances. Finally, we ran the experiment with scope 4, and ADUSA generated the counter example from the first instance. Figure 8 shows the data stored in the tables and the result of executing query `Q4`. Note how Oracle11g erroneously returns 12 as the query result instead of 4, the expected result when performing the natural join operations. To verify the result, we repeated the experiment with a previous version of Oracle, 10g. Our experiment showed that Oracle10g did in fact correctly compute the query result.

B. Experiments with MySQL

The aim of this experiment is to test ADUSA's ability in reproducing previously reported bugs in database management systems. We selected MySQL as a subject DBMS, which has an easily accessible bug repository.

We mined the bug repository and identified a bug in MySQL4.0 (bug number = 13371) with an SQL query that operates on a single table with a predicate that manipulates integer variables. The database schema and the test query are described below:

```
CREATE TABLE keytest (
    id int(10) NOT NULL default '1',
    PRIMARY KEY (id)
)

SELECT *
FROM keytest
WHERE (id=4 OR (id>=1 AND id<=3) OR (id>=2 AND id<=5))
```

We ran ADUSA using the above schema and test query on MySql4.0. In this experiment, we configured ADUSA not to terminate after the first inconsistency is detected. Using 4 integer objects, ADUSA generates 826 test cases

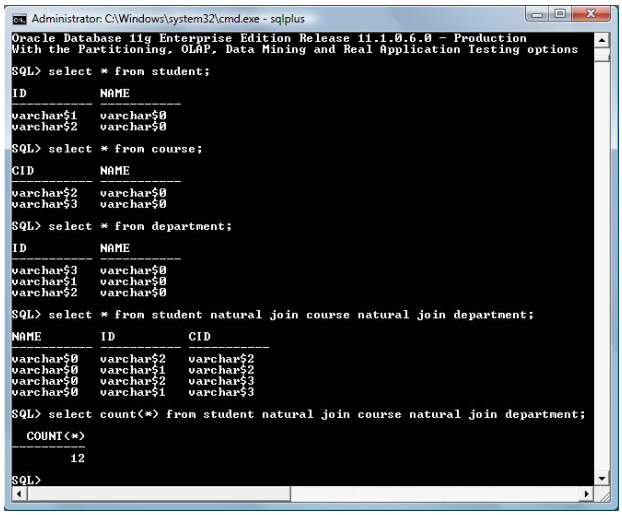

Fig. 8. Snapshot for running Oracle11g on the ADUSA generated counter example. The query execution returns 12 instead of 4.

in less than 10 seconds on a Pentium M machine with 512MB of memory. Out of these instances, 273 resulted in an inconsistency in the query execution, and 553 resulted in a valid query execution.

To study the efficiency of the automation process, we concurrently tried reproducing the bug without referring to the ADUSA generated counter example. It took us around 1 hour to (1) develop a suite of 50 test cases, (2) write test scripts to populate the database with the test data, and (3) write test scripts with the expected query result to verify the execution output. Out of these test cases, only two instances where able to reproduce the bug.

C. Experiments with HSQLDB

In this experiment, we use fault injection to add bugs in an open source DBMS, HSQLDB, and we check whether ADUSA was able to detect the injected bugs. The aim of this experiment is to perform a sanity check that ADUSA is not reporting invalid bug reports for a given test query.

We start our experiment by running a test query on HSQLDB before injecting a bug. We use ADUSA to verify the consistency of the query execution result. We then inject a bug in the implementation of the internal data structures that maintain the data in HSQLDB. Finally we run the ADUSA analysis on the same test query and we check whether ADUSA reports a counter example.

The test query used in this example is displayed below:

```
SELECT sid
FROM student s, course c
WHERE s.id = c.cid
```

We first ran ADUSA using the above test query on HSQLDB. ADUSA verified 1,108 instances in 10 seconds

for 4 varchars without reporting any inconsistency in the query execution results.

We then injected a bug in the implementation of the internal HSQLDB search. HSQLDB maintains the indices of the database records in an internal data structure that implements a balanced binary search tree. We modified the search routines used in HSQLDB by ignoring all the elements that reside in the right branch of the tree. However, we did not modify the code for inserting the indexes in the tree. Thus, indexes are still inserted correctly, yet only part of the inserted data is accessible during the search.

We ran ADUSA on the faulty implementation of HSQLDB. For the same test query, ADUSA reported 499 counter examples out of a set of 1,108 test databases. We point out that almost half of the generated test cases did not detect the injected bug. This shows the importance of exhaustive enumeration of test cases (even with small scopes), as it increases the chance of detecting insidious bugs.

V. Discussion

We next discuss some characteristics and limitations of our approach and present promising future directions.

A. Scalability of ADUSA

In Section IV, we presented several experiments for generating test database instances as well as verifying the results of a query execution on database management systems. To run an experiment, in addition to a test query and a database schema, we provided a scope that specifies the number of elements for each datatype declared in the database schema.

The values that we used for the scope ranged between 2 to 4 elements for each datatype. Since ADUSA uses the Alloy Analyzer to enumerate all the test databases that satisfy the Alloy specification, it cannot handle scopes with much larger sizes as it is infeasible to enumerate all instances (especially for large databases). This limits the application of ADUSA to bounded exhaustive testing [18] where a system under test is verified for all inputs with a small sizes.

While this limitation prevents ADUSA from generating large databases, the query-aware generation still enables generating a collection of databases that are large enough to cover a range of interesting scenarios for testing SQL queries. The case studies presented in this example illustrate how by covering the different combinations of tables with small number of records ADUSA was still able to discover new bugs in database management systems.

B. Handling more complex data types

The Alloy language provides a built in notation for the integers datatype which simplifies the analysis of SQL queries that manipulate integers. Alloy also provides set operations that enable handling complex relational operations, similar to those used in databases. Alloy types however are uninterpreted, and Alloy does not provide a representation for complex types such as, varchar, date, and time datatypes.

In this paper, we described how ADUSA models varchar data types in Alloy, which enables performing equality comparisons on strings in Alloy. More complex analysis, e.g. regular expression, is currently not possible in ADUSA. We are currently developing a string representation by modeling an automata class in Alloy, this enables extending ADUSA to support a larger SQL grammar.

We are also investigating the use of integer and string constraint solvers within ADUSA. While Alloy provides the constructs for modeling relations, a constraint solver provides a tool for handling operations on data.

VI. Related Work

This section compares ADUSA to prior work on specification testing using Alloy and database testing.

The use of Alloy for modeling and analyzing software systems is not new. Alloy has been previously used to analyze software systems [15], [17]. These techniques involve modeling both the program inputs as well as the computations in Alloy. Alloy has been also used for specification based testing. The TestEra [16], [18] framework for specification based testing generates test cases from Alloy specifications of data structures.

ADUSA uses a similar approach to that of TestEra in using Alloy to specify the input and for checking the correctness of the output. ADUSA, however, differs from Testra in two key ways. (1) TestEra is specialized for testing programs that take linked data structures as inputs, whereas ADUSA targets testing database management systems. (2) TestEra requires the programmer to learn a new specification language in order to specify the input. ADUSA, on the other hand, only requires the input described in SQL (the language of the application under test) and automatically generates the Alloy specifications.

The use of Alloy and the Alloy analyzer in ADUSA enables the unification of two key approaches for database testing: test database generation and test oracle generation.

Test database generation is a well studied problem [2], [4], [7], [11], [22]. Several approaches perform data generation by analyzing a given database schema [6], [7]. Such approaches aim at generating large databases that are used as benchmarks for performing various analyses on databases.

Other database generation approaches [2], [3] take the database schema as well as a test query into consideration when generating data values. Such approaches aim at generating test databases that result in meaningful query

executions for testing DBMS [3] systems or applications that interact with databases [9], [25].

Our approach for data generation in ADUSA is query-aware and targets generating a large set of small databases for exhaustively testing a DBMS system. Unlike other approaches which use constraint solvers [3], or object modeling language [22], ADUSA uses the Alloy Analyzer which in turn uses SAT to generate its data. The use of Alloy enables specifying constraints on both the query as well as the results which enables more precise test input generation.

Several database testing approaches target transaction testing, i.e., checking the effect of executing a sequence of related SQL queries on a database. For example, a recent tool, AGENDA [8], uses state validation techniques to verify the consistency of the database after executing a transaction. Our approach in ADUSA does not target transaction testing. ADUSA primarily considers SQL selection statements which unlike transactions do no update the state of the database.

VII. Conclusion

We have presented a novel approach for automatic DBMS testing. Our approach generates both the input databases for a test case and a test oracle to verify the result of a test. To generate the inputs and the oracles our approach uses the Alloy Analyzer which is based on SAT. The key idea behind our approach is to systematically model a relational database using the Alloy relational specification, and then use the Alloy Analyzer to generate the test input as well as the test oracles.

We presented several case studies where we used our approach to detect bugs in both commercial and open-source DBMSs. We believe that automatically generating test inputs as well as test oracles in a unified framework is an important step towards fully automated database testing frameworks.

Acknowledgments

We thank Don Batory for pointing us to the buggy count function in Oracle and for useful feedback on the class project report of the first and third authors. We thank Zoran Martic and the anonymous referees for their detailed and helpful comments on the paper. This work was funded in part by grants #CCF-0702680 and #IIS-0438967 awarded by the National Science Foundation.

References

[1] MySql open source database. http://www.mysql.com/.

[2] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *ICDE*, pages 506–515. IEEE, 2007.

[3] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: generating query-aware test databases. In *SIGMOD Conference*, pages 341–352, 2007.

[4] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107. VLDB Endowment, 2005.

[5] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

[6] Datanamics. Db date generator. http://www.datanamic.com/datagenerator/index.html.

[7] IBM DB2. Test database generator. www.ibm.com/software/data/db2imstools/db2tools/db2tdbg/.

[8] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 78–87, 2005.

[9] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162, 2007.

[10] Larry Greenemeier, Elena Malykhina, and Paul McDougall. The high cost of data loss, March 2006. http://www.informationweek.com/showArticle.jhtml?articleID=183700367.

[11] Kenneth Houkjaer, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[12] Daniel Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, November 2000.

[13] Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), April 2002.

[14] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[15] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, August 2000.

[16] Sarfraz Khurshid and Darko Marinov. Checking Java implementation of a naming architecture using TestEra. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[17] Sarfraz Khurshid and Darko Marinov. Using TestEra to check the Intentional Naming System of Oxygen. In *MIT Student Oxygen Workshop*, Gloucester, MA, July 2001.

[18] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 2004.

[19] Oracle Corp. Oracle database engine. http://www.oracle.com/technology/software/products/database/index.html.

[20] Meikel Poess and Jr. John M. Stephens. Generating thousand benchmark queries in seconds. In *vldb'2004: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1045–1053, 2004.

[21] Donald R. Slutz. Massive stochastic testing of sql. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 618–622. Morgan Kaufmann, 1998.

[22] Yannis Smaragdakis, Christoph Csallner, and Ranjith Subramanian. Scalable automatic test data generation from modeling diagrams. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 4–13, New York, NY, USA, 2007. ACM.

[23] The hsqldb Development Group. HSQL database engine. http://www.hsqldb.org/.

[24] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. pages 632–647. 2007.

[25] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 102–111, New York, NY, USA, 2006. ACM.