An Intensional Approach to the Specification of Test Cases for Database Applications

David Willmor
School of Computer Science
University of Manchester
Oxford Road, Manchester, UK
d.willmor@cs.manchester.ac.uk

Suzanne M. Embury
School of Computer Science
University of Manchester
Oxford Road, Manchester, UK
s.m.embury@cs.manchester.ac.uk

ABSTRACT

When testing database applications, in addition to creating in-memory fixtures it is also necessary to create an initial database state that is appropriate for each test case. Current approaches either require exact database states to be specified in advance, or else generate a single initial state (under guidance from the user) that is intended to be suitable for execution of all test cases. The first method allows large test suites to be executed in batch, but requires considerable programmer effort to create the test cases (and to maintain them). The second method requires less programmer effort, but increases the likelihood that test cases will fail in non-fault situations, due to unexpected changes to the content of the database. In this paper, we propose a new approach in which the database states required for testing are specified intensionally, as constrained queries, that can be used to prepare the database for testing automatically. This technique overcomes the limitations of the other approaches, and does not appear to impose significant performance overheads.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Experimentation, Verification

Keywords

databases, software testing, database testing

1. INTRODUCTION

Modern information systems are typically organised as collections of independent application programs that communicate with one another by means of a central database. The database records the state of the organisation that the information system supports, while the application programs implement the business processes that manipulate the state. To take a simple but ubiquitous example, a database system might record details of customers, products and sales, while the application programs associated with it handle operations such as new product purchases and update of the product catalogue, as well as supporting decision making by generating reports regarding the most profitable product lines, names and addresses of loss-making customers, *etc.*

In order to test such application programs, it is necessary to create test fixtures that simulate the presence of the rest of the information system. Fixtures for traditional test cases typically consist of in-memory objects and data structures that provide the inputs to the program being tested. This kind of fixture is also needed when testing database applications (especially when performing unit testing); however, since it is unrealistic (and often incorrect) to execute test cases against an empty database, we need to create additional fixture elements within the database itself.

Current practice in the software industry is to maintain one or more test databases that can be used for testing individual programs. These databases can be artificially generated (e.g., using tools such as DBMonster[1] and DataFactory[2]) or they may be subsets of the live database, taken as a snapshot at some recent point in time. Copies of the live data sets have the advantage that they are more likely to be representative of the patterns of data encountered in practice, while artificial data sets have the advantage that they can be made to embody specific characteristics (such as particular data skew patterns or volumes), which may be useful for load and stress testing.

Both approaches, however, suffer from several disadvantages. The most significant problem occurs when none of the available test databases are suitable starting points for a particular test case. For example, suppose a particular test case executes a program which purges inactive customers, with the aim of verifying that the business rule forbidding deletion of customers with negative balances is correctly enforced. If none of the test databases contains any inactive customers with negative balances, then the test case cannot be executed successfully. For a one-off test run, testing personnel can choose a database that is close to what is required, and manually update it so that it is suitable for use with the test case. But if a complete test suite is to be executed (possibly including test cases which themselves make modifications to the database state) then in the worst case

[1]`http://DBMonster.kernelpanic.pl`
[2]`http://www.quest.com/datafactory`

this manual intervention will be required in between every test case execution. This is clearly undesirable if test suites are large or time-consuming to execute, or if the test suite is to be run in batch (as in the case of overnight regression testing, for example).

Current research in testing for database systems proposes two approaches to this problem. One of these is to include within the test case description a full (*extensional*) specification of the database state against which it is to be run (and of the database state that should be produced if the test has executed successfully) [13, 14]. This solution is exemplified by *DBUnit*[3], an extension of the JUnit testing framework[4] that is designed for testing database applications written in Java. Each DBUnit test case is accompanied by an XML file describing the data set required for the test. Before each test run, DBUnit clears the database state and inserts the data described by the XML file.

This approach has the advantage of simplicity, but it places a considerable burden on testing personnel, especially when complex database states are required. It is also inefficient, since the database must be continually destroyed and recreated between tests, even when significant parts of the database might have been reused by the succeeding tests. Moreover, maintenance of a large suite of such tests is extremely challenging, since any small change to the database schema may require corresponding changes to many test cases.

The second approach that has been explored in the literature is more efficient in that it requires the creation of only one database state per test suite (rather than one per test case). It is exemplified by the AGENDA database testing toolkit [6, 7], which can automatically generate a database state given information about the schema, some data generation functions for individual attributes and some user-selected heuristics describing the kind of database state required. The AGENDA tool also generates test cases from a simple analysis of the program being verified. The user must then add preconditions to each test case that are checked just before it is executed and that will prevent a case from being executed against an inappropriate database state. This approach successfully relieves the user of the need to specify complete database states in full detail, but at a cost. The user must accept that some of the test cases may not be executed because the database state fails the precondition, even when it would require only a small change to bring the database into a suitable state for the test. Since only one database state is created per test suite, this problem of failed tests is likely to become more severe as the size of the test suite grows. There is also a potential inefficiency involved in generating test descriptions and inputs, and in creating the additional log tables and constraints/triggers needed by the AGENDA tool, for test cases that are not in fact going to be executed.

Ideally, we would prefer to be able to combine the advantages of both these approaches, to give a form of database test case that is quick and natural to specify, and which maximises the number of cases within the suite that can be executed while minimising the number of full test databases that need to be maintained. Our thesis is that this can be achieved by allowing testing personnel to describe the database states involved in their test cases *intensionally*, in the form of declarative conditions that the input database must satisfy, and by providing a testing harness that can automatically adjust the input database so that the test conditions are satisfied [19].

[3]`http://www.dbunit.org`
[4]`http://www.junit.org`

In this paper, we present a language for specifying such intensional database tests, and describe its semantics and operational behaviour (Section 2). We present an algorithm for automatically modifying database states so that test preconditions are satisfied (Section 3), thus ensuring that all test cases can be executed without requiring any human intervention. We further describe how we have extended the JUnit testing framework to allow intensional database tests to be specified and executed in practice (Section 4). Finally, we present the results of an evaluation of the performance of the techniques (Section 5) and conclude (Section 6).

2. SPECIFYING INTENSIONAL TESTS

A conventional test case is typically modelled as a triple $< p, i, o >$, which denotes a test that executes program p with inputs (e.g., parameters) denoted by i. If no faults are encountered during the test execution, the output that will be produced is o. In the case of test cases for database applications, we must add two further elements—the specification of the database state against which p is to be executed, and some statement of the database state that should result from the execution of p if it is operating correctly according to its specification.

For example, consider the example program mentioned in Section 1 that prunes inactive customer details from the database. For this test case, we require a database state that contains at least one inactive customer. This could easily be stated as a predicate logic condition over the database, assuming the obvious mapping between stored relations and predicates, e.g.:

$$\begin{gathered}(\exists custNo, lastOrderOn, a, b, c)\\ customer(custNo, a, b, c, lastOrderOn) \wedge\\ lastOrderOn < today - 90\end{gathered}$$

The program in question does not access any parts of the database other than the `customer` table. Therefore, we do not care what values the other tables contain and need not mention them in the intensional specification of the test.

This approach works equally well for observing the results of the test. For example, when testing the customer pruning behaviour, we might require that no inactive customer with a non-negative balance should exist in the database after the test:

$$\begin{gathered}\neg((\exists custNum, lastOrderDate, a, b, c)\\ customer(custNum, a, bal, c, lastOrderDate) \wedge\\ lastOrderDate < today - 90 \wedge bal > 0)\end{gathered}$$

Effectively, the test case describes a set of valid (i.e., fault-free) state transition for the database, as a classic pre/post-condition pair.

This first-order-logic style of database specification does not work so well when we consider the testing problem in more depth, however. The problem is that we need to do more than test the input database for compliance with the requirements of the test case; we also need to extract information from it to be used to instantiate other elements of the test case. For example, suppose we wish to test a program that deletes details of individual customers. Such programs typically require some input from the user, identifying the specific customer record that is to be deleted (e.g., by supplying the relevant customer code as a parameter). This could be achieved by requiring the tester to embed the customer code into the test case elements, as literal values. Alternatively, we could search for a suitable customer that already exists in the database, using a standard database query, and use the values from that in specifying the inputs for the test case. This would minimise the amount of work required to prepare the database for test execution (since we would be using data already present in the database), and it would also mean that test cases can be written very quickly, since the user does not need to specify every last detail of the data to be used.

Under this approach, the specification of the input database state now has a dual role: it must state the condition that determines whether the database state is suitable for execution of the test case *and* it must also return bindings for the free variables that appear in the remaining components of the test case. For the latter purpose, we would prefer to use a straightforward query language, while for the former we require the ability to place conditions on the data. With a simple extension of a standard query language such as SQL, we can combine both these purposes in a single statement. For example, the following statement:

```
ANY :cn GENERATED BY
  SELECT custNo FROM customer
  WHERE lastOrderDate < today() - 90
    AND balance < 0
```

retrieves the customer code of some record that meets the given conditions (an inactive customer with negative balance) from the database, and binds it to the variable `:cn`. It also places a cardinality constraint on the result of the query, that at least one such binding must exist (implied by the use of the keyword `ANY`).

The variable `:cn` can then be used to specify other elements of the test case. The obvious usage in this example is in specifying the inputs to the program being tested, but it can also be used in describing the expected outputs of the program. In this example test case, the correct behaviour of the `DeleteCustomer` program is to reject the deletion of `:cn`, since customers with a negative balance cannot be purged from the database. We might therefore give the following specification of the desired output database state:

```
AT LEAST 1 :cn2 GENERATED BY
  SELECT custNo FROM customer
  WHERE custNo = :cn
```

Of course, not all test cases are best specified in terms of values retrieved from the database. For example, suppose that we wish to write test cases for a program that adds new customers to the database. The inputs to this program are the details of the new customer, and the precondition for one particular test case states that no customer should exist that has the same customer code as that of the customer being created. We cannot retrieve the customer details from the database in this case, as they have not yet been stored in it. Again, we could force the user to include the required values as literals in the test case, but ideally we would like to give

```
<CONDITION> ::= <TYPE> <BINDINGLIST>
                GENERATED BY <SELECT>
<TYPE> ::= ANY | NO | AT LEAST <i> |
           AT MOST <i> | EXACTLY <i> |
           ALL | FIRST
<i>    ::= {0-9}
<BINDINGLIST>
       ::=  <BINDING> { ',' <BINDINGLIST> }
<BINDING> ::= {A-Z | a-z}
<SELECT>  ::= ...
```

Figure 1: Simplified BNF Grammar for SQL Extensions

more support to the process of test case generation. One way to achieve this is to allow user-defined data generator functions to be incorporated within queries as though they were relations. For example, the following expression states our requirements for this test case, while also binding the variables needed for input to the program:

```
ANY :cn, :name, :addr, :bal GENERATED BY
  SELECT gc.custno, gc.name, gc.addr, 0
  FROM genCustomerDetails() AS gc
  WHERE gc.custno NOT IN (
        SELECT custno
        FROM customer
        WHERE balance > 0)
```

Here, the data generator function `getCustomerDetails()` is used as if it were a normal relation, whereas in fact the results it returns are computed on the fly. In fact, several of the main commercial database management systems already allow user-defined functions to be embedded in queries in this way, so this does not require a further extension of SQL. Figure 1 shows the minimal extensions that are needed to support all the kinds of constrained query shown above using the SQL99 standard [17].

2.1 Test Case Semantics

Clearly, the semantics of these intensional database test cases is more complex than for traditional extensional tests. However, we can define their semantics formally in terms of a mapping from intensional tests to sets of equivalent extensional database test cases. We first present a formal definition of the structure of our intensional test cases:

DEFINITION 1. *An intensional database test case is a quintuple* $< p, i, DB_i, o, DB_o >$*, where:*

- p *is the program to be executed in the test,*
- i *is a tuple of* n *variables and literals that describes the inputs to be given to program* p*, where* n *is the number of parameters expected by* p*,*
- DB_i *is a set of constrained queries that together specify the initial database state.*
- o *is a tuple of* m *variables and literal that describes the expected outputs from the program* p*.*
- DB_o *is a set of constrained queries that together specify the conditions that must hold in the database state after execution of* p *if no fault has been encountered.*

A constrained query has the form $< Q, min, max, vars >$, where Q is a standard relational algebra query, *min* and *max* describe the constraints on the cardinality of the query result set, and *vars* is the list of variables bound by the query result.

A database test case is well-formed for use with a particular database schema Σ iff:

- for every variable v that occurs free in i, DB_i, o and DB_o, there exists a query in DB_i that provides a binding for v,
- for every query $< q, n, m, vs >$ in $DB_i \cup DB_o$, q is a well-formed query over Σ that returns k-tuples, where $|vs| = k$, and
- there are no circular variable dependencies amongst the queries in DB_i.

We can now define a semantics for the intensional database test cases as follows. Every intensional test case is equivalent to a set of *extensional test cases.* An extensional test case defines a specific test run, in terms of actual inputs and outputs, rather than expressions denoting sets of inputs and outputs. The set of all possible extensional test cases is given by:

$$\mathcal{P} \times \mathcal{L}^n \times \mathcal{DB} \times \mathcal{L} \times \mathcal{DB}$$

where $\mathcal{P}$ is the set of all programs, $\mathcal{L}$ is the set of all literals, $\mathcal{L}^n$ is the set of all n-tuples formed from $\mathcal{L}$ and $\mathcal{DB}$ is the set of all database states (relative to all schemas)[5]. The components of each extensional test are the program to be tested, the input values, the initial database state, the expected output and the expected final database state, respectively.

An intensional test case is effectively a shorthand expression for a set of extensional test cases that are all derived from the same equivalence partition of the test case inputs. An intensional database test $< p, i, DB_i, o, DB_o >$, where $DB_i = \{< q_i, n_i, m_i, v_i >\}$ and $DB_o = \{< q_o, n_o, m_o, v_o >\}$, is equivalent to the following set of extensional tests:

$$\begin{aligned} \{< p, i[v_i/v], db_i, o[v_i/v], db_o > |\ & \\ db_i \in \mathcal{DB} \wedge & \\ (n_i \leq |q_i(db_i)| \leq m_i) \wedge & \\ v \in q_i(db_i) \wedge & \\ db_o \in \mathcal{DB} \wedge & \\ (n_o \leq |(q_o[v_i/v])(db_o)| \leq m_o)\} & \end{aligned}$$

We use the notation $exp[\theta_1/\theta_2]$ to express the substitution of the values in θ_1 by the corresponding values in θ_2 whereever they occur in *exp*. Therefore, this expression denotes the set of extensional tests where the input database satisfies the constraints imposed by the initial constrained query, and where the bindings from execution of that query (here expressed as the tuple of variables v) are substituted into the expressions defining the inputs, expected output and expected final database state before they too are evaluated[6].

The idea underlying this notion of an intensional test is that when any of its corresponding extensional sets are executed, the intensional test is itself deemed to have been executed. Thus, the use of intensional tests allows much greater freedom at test execution time, since we may choose any of the possible extensional tests, depending on which is closest to our starting environment. In the next section, we will consider the practical ramifications of this approach to testing, and describe how the semantics just described can be implemented in practice.

3. DATABASE PREPARATION

The execution of an intensional database test case consists of three distinct phases: 1) preparation of the environment for test execution; 2) execution of the test with the prepared inputs; and 3) capture and storage of the results, for later analysis. Since all the work of finding bindings for the variables in the test case specification is done in the preparation phase, the final two phases are straightforward and differ little from standard testing procedures. When program execution is complete, the constrained query that determines whether the test has been successful or not is evaluated against the database, and the output from the program is checked against what is expected. In the case of test failure, the details of the actual extensional test that was executed are recorded, for diagnosis purposes.

The first phase, however, is more complex. If we were content to execute only those test cases which happen to be suitable for use with the initial database state, then the preparation phase would simply be a matter of executing the input constrained queries against the database and, if they are all successful, using the bindings thus produced to instantiate the remaining components of the test case. However, thanks to the declarative nature of our test case specifications, the testing framework can be pro-active in cases where the given database is not suitable for use by the test case, and can automatically generate a sequence of updates that will cause the constrained queries to produce the required number of bindings.

In fact, this problem is similar (though not identical) to one that has been studied by the database and artificial intelligence communities for many years. It is known variously as the *view update* problem [9], the *knowledge base update* problem [12], and the *transaction repair* problem [10]. Many database systems have the capability to define views on top of the basic database. A view is a kind of virtual relation. To the user, it appears to be a normal relation, but it contains no stored data. Instead, the contents of the view are defined by a expression over other relations, and attempts to retrieve data from the view are converted into queries over these relations. To take a simple example for illustration, we might create a view called `Debtors` which appears to be a relation of the same name containing all customers with a negative balance. Attempts to retrieve `Debtors` is

[5]For simplicity of presentation, we assume that all programs require the same number of inputs (n). In practice, n can be the largest number of inputs required by any program, and the unused values can be filled with nulls.

[6]For simplicity of presentation, we assume here that there is only one query in each of DB_i and DB_o. In practice, it may be necessary to include several queries, each producing different bindings and imposing different cardinality constraints. In this case, the constraints must be conjoined, and the full set of bindings can be retrieved by performing a natural join of all the queries, with join condition *true*.

converted into a query against the customer table with an added constraint on the balance.

If views are truly to act as normal relations then it should be possible to update them as well query them. But what does it mean to update a virtual relation? In this case, the view update must be converted into a sequence of updates on the stored relations that will cause the desired change in the contents of the view itself. This is a non-trivial problem for realistic view languages, and becomes even more difficult when we move into the context of knowledge bases, where virtual relations can be defined using rules over other relations, and when we add integrity constraints that must be maintained by all updates [1, 2, 3, 4, 5, 8, 11].

Only in very narrow circumstances does a view update have a single translation into real updates [15, 18]. Various heuristics for selecting from amongst the possible translations have been proposed (of which the most common is to choose the update that results in the smallest change to the existing data set [2]), but in real applications user input is needed in order to identify the translation that corresponds most closely to the real world state that the database should reflect [10].

In the case of intensional database tests, we have a query (the constrained query that describes our requirements for the test) that does not produce the correct number of answers when executed against the test database. We need to find a sequence of updates to the base data that will cause our query to produce the number of answers we need. However, in this case, there is no requirement to find the set of updates that matches the state of reality — any sensible update that satisfies the query conditions will be acceptable. This simplifies the problem considerably, removing the need for complex search procedures and for any user input.

3.1 The Preparation Algorithm

One of the advantages of using a query-based language for test specification (as opposed to a predicate calculus-based language) is that we can make use of a very common and easy-to-analyse internal form for (relational) database queries, called relational algebra. This form provides a small number of operations on relations that can be combined to form complex queries. For example, the three most basic (and useful) relational algebra operators are:

- The projection operator, $\pi_{Atts}R$, which creates a relation from R by deleting all attributes not in $Atts$. For example, $\pi_{[Country]}\,Customer$ produces a relation that contains just the countries that appear in the `Customer` relation.
- The selection operator, $\sigma_c R$, which creates a relation that contains all the rows from relation R that satisfy the condition c. For example, $\sigma_{bal<0}\,Customer$ returns a relation containing details of all customers with negative balances.
- The join operator, $R \bowtie_c S$, which creates a relation containing rows from the cross product of R and S that satisfy the join condition c. The query $Debtor \bowtie_{dNo=iNo} Inactive$ returns details of all debtors who are also inactive.

Since the result of each relational algebra operator is itself a relation, together they form a closed algebra. This means that we can form arbitrarily complex queries by applying operators to the results of other operators. For example, a query which retrieves the customer number of all customers with a negative balance would be written as:

$$\pi_{[custNo]}(\sigma_{balance<0}\,Customer)$$

A common way to visualise such expressions is as a tree of operators. The tree for the above query is shown in Figure 2.

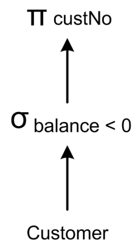

Figure 2: Relational Algebra Tree for Negative Balance Query.

Our algorithm for preparing a database for testing is based around this notion of a relational algebra tree. We take the cardinality constraints from the test specification, and push them down through the nodes of the input database query tree, collecting up additional conditions as we go. When we reach a leaf node (i.e. a base relation), we make updates to the database so that the pushed-down constraints are satisfied for that relation.

At each stage, we collect up the different kinds of constraint and push them further down into the tree. These constraint types are:

- *Min* and *Max*, the upper and lower bounds on the desired cardinality of the result set.
- *SelC*, the selection conditions on the relations that we are interested in.
- *UAtts*, the collection of attributes that are used in the constrained query, and that must be populated in any new data that we insert.

We also build up a collection of queries that describe the data that has been prepared for testing so far, as we progress through the tree. We call these queries "bindings" (Bgs), since they give us values for the variables that occur within the selection and join conditions. At each stage, the bindings should contain one query for each leaf node that has so far been prepared.

It is easiest to see how this works by considering a simple example, such as that shown in Figure 2. Let us assume we have a constrained query that requires at least one customer with negative balance to exist, and that our database does not currently contain any such customers. We begin at the root node of the tree, with only the cardinality constraints extracted from the test specification:

$$Min = 1,\ Max = null,\ SelC = true,$$
$$UAtts = \emptyset,\ Bgs = \emptyset$$

The top node is a projection operator. Projection does not affect the cardinality of the result set, nor impose any conditions, but it does tell us something about the attributes used

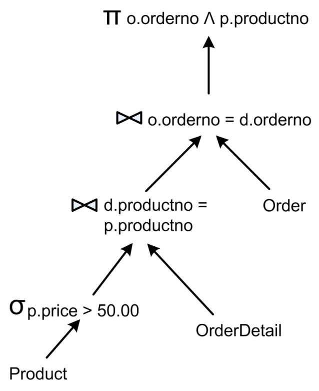

Figure 3: Relational Algebra Tree Showing Multiple Joins

by the query. We therefore add the projection attributes to *UAtts* and push the constraints down to the next node:

$$Min = 1,\ Max = null,\ SelC = true,$$
$$UAtts = \{custNo\},\ Bgs = \emptyset$$

Next we must deal with the selection node. Selection nodes reduce the cardinality of their input, so we need to push down the selection conditions to ensure that any updates we may make affect the correct tuples. We also need to add any attributes appearing in the selection condition to *UAtts*:

$$Min = 1,\ Max = null,\ SelC = balance < 0,$$
$$UAtts = \{custNo,\ balance\},\ Bgs = \emptyset$$

The final node is the leaf node, representing the `Customer` relation. We construct a query from the conditions on that relation and execute it, to find out how many answers are currently in the database. In this case, there are none, so we need to insert a new `Customer` record with at least the `custNo` and `balance` attributes populated, and with a negative balance. If there are any integrity constraints on this relation, then we need to make sure they are also satisfied by the new data.

We use the DBMonster data generator mentioned earlier to create the new data. It allows generation functions to be specified for attributes, and additional constraints to be placed on them. It will also maintain primary key, foreign key, non-null and domain constraints if configured appropriately using the information present in the pushed-down constraints.

Of course, this is a very simple example. In general, we can expect to have to deal with more complicated queries involving several joins, such as that shown in Figure 3. This relational algebra tree is equivalent to the following constrained query:

```
ANY :orderNo, :productNo GENERATED BY
SELECT o.orderno, p.productno
FROM Order o, Orderdetail d, Product p
WHERE o.orderno = d.orderno AND
      d.productno = p.productno AND
      p.price > 50
```

which requires that at least one order must exist that involves the purchase of at least one product that costs more than £50. Joins complicate the process of preparing the database, because they introduce dependencies between the updates that take place at different leaf nodes. For example, imagine that we have processed the tree shown in Figure 3 as far as the leaf node representing the `OrderDetail` relation. Join operators further constrain the selection condition (by conjoining in their join condition), but add no other constraints. So, by the time we reach this leaf node, *SelC* will have been set to:

$$o.orderno = d.orderno \wedge d.productno = p.productno$$

We need to find out whether a suitable `OrderDetail` record exists within the database. However, in order to do this, we need to know something about what preparation actions were performed when the `Product` leaf node was processed. Maybe there were already plenty of £50-plus products in the catalogue, or maybe there were none and one had to be created. How is this information passed through to the `OrderDetail` node so that the correct tuple can be identified or created?

In the current version of our algorithm, we have chosen to use the database itself to communicate these values. If there are many suitable `Product` records, then we can find one by querying the database directly once again. If a new product had to be created, then it will now be present in the database, so we can still retrieve it by querying. The information needed to construct these queries is present in the selection conditions that have been considered during the processing of the relational algebra tree up to this point. For example, in order to search for an `OrderDetail` tuple that is connected to a suitable `Product`, we need to issue the following query:

```
SELECT d.* FROM OrderDetail d, Product p
WHERE d.productno = p.productno AND
      p.price > 50
```

This query cannot be constructed from only the constraints pushed-down from the parent nodes of the leaf node; instead, we need to collect up the constraints imposed by all nodes visited before the current node, so that they are available for query formation. This is done using the *Bgs* data structure mentioned earlier.

Figure 4 presents the complete algorithm, showing the behaviour required for each different type of operator. The algorithm is presented as a side-effecting function which takes the constrained query that is to be satisfied by the database, and a set of initial conditions that state the required cardinality bounds and initialise *SelC* to *true*, *UAtts* to $\emptyset$ and *Bgs* to $\emptyset$. The function returns a set of bindings, but these are discarded. The main task of the algorithm is carried out by the side-effecting updates that occur when leaf nodes are processed.

4. DOT-UNIT TESTING FRAMEWORK

The intensional database test language and accompanying preparation algorithm have been implemented within a testing tool, called *DOT-Unit*. This tool is part of a larger Data-Oriented Testing[7] framework that is under development at the University of Manchester [20]. DOT-Unit has been implemented as an extension to the JUnit testing framework

[7]`http://www.cs.man.ac.uk/~willmord/dot/`

Projection operator

```
prepare($\pi_{Atts}Q$, Min, Max, UAtts, SelC, Bgs)
    = prepare($Q$, Min, Max, UAtts $\cup$ Atts, SelC, Bgs)
```

Selection operator

```
prepare($\sigma_c Q$, Min, Max, UAtts, SelC, Bgs)
    = prepare($Q$, Min, Max, UAtts, SelC $\wedge$ $c$, Bgs)
```

Join operator

```
prepare($Q_1 \bowtie_{jc} Q_2$, Min, Max, UAtts, SelC, Bgs)
    = prepare($Q_2$, Min, Max, UAtts, SelC $\wedge$ $jc$,
          prepare($Q_1$, Min, Max, UAtts, SelC, Bgs))
```

Relation (leaf node)

```
prepare($Rasv$, Min, Max, UAtts, SelC, Bgs)
    $Q$ = bindingQuery($v$, SelC, Bgs)

    Execute $Q$ to produce result set $RS$

    if $|RS|$ < Min then
        Invoke DBMonster to create (Min - $|RS|$) more
        instances of $R$ that satisfy the conditions in $Q$
    else if $|RS|$ > Max then
        Delete the first ($|RS|$ - Max) tuples in $RS$
    else
        No preparation updates needed

    return (Bgs $\cup$ binding($v$, $Q$))
```

Figure 4: The Database Preparation Algorithm

for the unit testing of Java applications [16]. We have subclassed the standard JUnit `TestCase` class, to create a dedicated `DatabaseTestCase` class for specifying and managing intensional database tests. `DatabaseTestCase` provides facilities for specifying pre-conditions on database state, generating and manipulating the bindings that are produced by such pre-conditions, and evaluating post-conditions on the database state after the test has been completed. The standard JUnit methods for determining the results of test execution on the in-memory fixture can also be used.

Figure 5 shows an example DatabaseTestCase that includes two individual tests. The first verifies that when a customer with a non-negative balance is deleted, all customers with that customer number really do disappear from the database. The second uses a data generation function to propose attribute values for a new customer record (including a unique customer number), and checks that after the program has executed only one customer with the generated customer number exists.

We use a prefixed colon to indicate variables that are shared amongst the test components — a notation that will be familiar to many database programmers, since it is commonly used in various forms of embedded SQL. The shared variables acquire their values when the test harness evaluates the precondition (and performs any necessary database preparation steps). These values can then be accessed using the `binding` method, and can be used in arbitrarily complex assert conditions, as well as in instantiating the post-condition query.

One of the main advantages of using the JUnit framework as the basis for the implementation of DOT-Unit is that it allows us to integrate our tool seamlessly into existing development environments, such as Eclipse[8]. Thus, DOT-Unit tests are executed in exactly the same way as a standard JUnit test case, and the results are displayed using the same interface components. This allows testing of database and non-database components to be interleaved in a convenient and natural manner.

[8]`http://www.eclipse.org`

5. EVALUATION

The practicality of this intensional test case approach depends largely on the performance overhead imposed by the database preparation algorithm. If the time required to execute each individual test case is significantly higher using our approach than with DBUnit, say, then fewer tests will be able to be executed in the time available and the benefits of faster test development and fewer spurious test failures will be negated.

To gain a handle on the degree of performance overhead to be expected from DOT-Unit, we made use of an existing extensional DB test suite that we created for earlier work [20]. This suite was designed for *mp3cd browser*[9], an open-source Java/JDBC program that stories information about mp3 files in a MySQL 5.0 database[10]. The schema of the database consists of 6 relations with 22 attributes, 7 primary key constraints and 6 foreign key constraints. We created an equivalent intensional test suite, consisting of 20 test cases, from the extensional suite by converting each test case into DOT-Unit pre- and post-conditions. We also replaced each hard-coded test parameter in the original tests into constrained query bindings.

We wanted to investigate two specific aspects of the performance of DOT-Unit. First, we wanted to compare its performance with that of DBUnit over the equivalent test cases as the database size grows. Second, we wanted to gain some idea of what aspects of DB preparation and testing were dominating the performance of DOT-Unit. The results of the experiments we performed are presented below. All experiments were run on a Pentium-M 2.0GHz machine, with 1Gb RAM, running Ubuntu Linux.

5.1 Comparison with DBUnit

At first sight, the extensional approach, as exemplified by DBUnit, would seem to be the more efficient method of the two, as the testing harness does not need to spend any time figuring out what updates need to be made prior to each test—it only needs to execute them. This does

[9]`http://mp3cdbrowser.sourceforge.net/mp3cd/`
[10]`http://www.mysql.com`

```
public class ProgramTest extends DatabaseTestCase {
     public void testDeleteCustomer() {
          preCondition("ANY :cn GENERATED BY SELECT custNo FROM customer WHERE balance > 0;");
          Program p = new Program();
          p.deleteCustomer(binding(":cn"));
          postCondition("NO :cn2 GENERATED BY SELECT custno FROM customer WHERE custNo = :cn;");
     }
     public void testNewCustomer() {
          preCondition("ANY :cn, :name, :addr GENERATED BY SELECT gc.custNo, gc.name, gc.addr FROM
               genCustomerDetails() AS gc WHERE gc.custNo NOT IN (SELECT custNo FROM customer);");
          Program p = new Program();
          boolean b = p.newCustomer(binding(":cn"), binding(":name"), binding(":addr"));
          assertTrue(b);
          postCondition("EXACTLY 1 :cn, :name, :addr GENERATED BY SELECT custno, name, addr
               FROM customer;");
     }
}
```

Figure 5: Example DOT-Unit Test Case

not happen by accident, but because a human programmer has spent time earlier, deciding exactly what the database should look like for each test case. However, when writing DBUnit tests, it is common to try to reuse database descriptions for multiple test cases where possible, to reduce the amount of programming and maintenance time. In this case, some redundant updates will be made before each test case - updates that our extensional approach will not bother to make. It is also the case that DBUnit makes its updates blindly, whether they are needed or not, whereas the intensional approach will be able to reuse much of the existing database state for each new test case.

Given this, it seems likely that the performance of DBUnit will be better when the database state required for each test case is relatively small, but that the situation will be reversed when the database state grows much larger. In order to gauge the point at which this change occurs, we ran our two test suites (extensional and intensional) with databases of varying sizes, and measured the execution time taken to execute the whole test suite.

In each case, we generated initial database states of varying sizes at random - either populating the database directly (for the intensional test cases) or generating XML descriptions of the required state (for the extensional test cases). The results are shown in Figure 6.

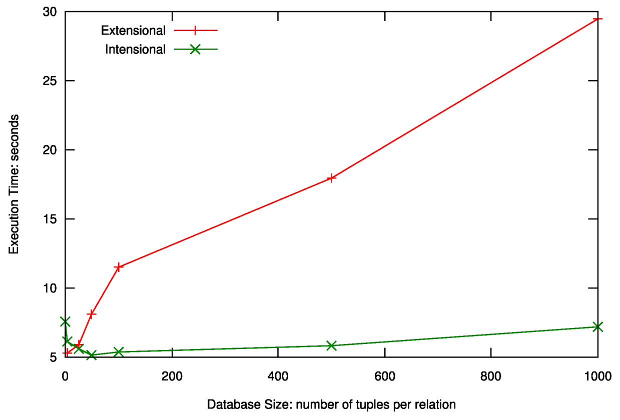

Figure 6: Comparison of Approaches as DB Size Increases

To our surprise, although the performance of DOT-Unit was initially worse than that of DBUnit, it overtook its competitor at a comparatively small database size of around 20 tuples per relation. Obviously, this experiment is a little unfair to DBUnit, since programmers are unlikely to create database descriptions consisting of 1000s of tuples per relation. However, tests of this scale will be needed at some point in the development cycle, in order to verify the behaviour of the system on more realistic data sets.

In order to assess the behaviour of DOT-Unit more precisely, consider the graph in Figure 7, which shows the results at small databases sizes in more detail. It can be observed that the performance of DOT-Unit first improves and then begins to degrade again at a database size of around 50 tuples per relation.

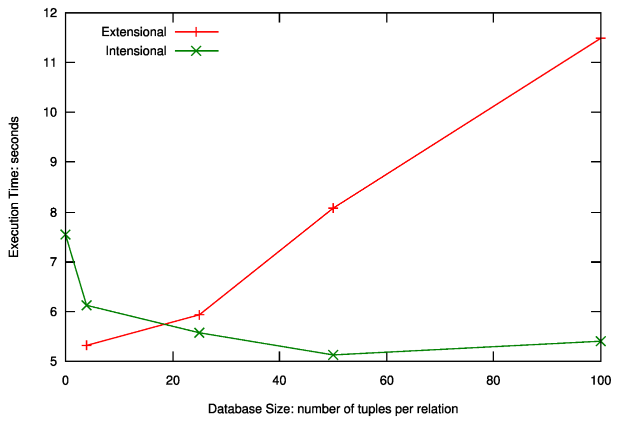

Figure 7: Detailed Comparison of Approaches

One possible explanation for this initial improvement in performance is that, as the database size rises, so does the probability that the data needed for the test case is already present in the database. For the very small states, a lot of preparation work is required to create the needed data, whereas less work is needed for a more fully populated database. As the database size increases further, however, the costs of making the queries needed to test the preconditions and formulate the preparation updates rises, pushing up the time required for the entire preparation step. This behaviour may be a peculiarity of the particular test suite used, of course, and further, more extensive studies will be required in order to completely characterise the performance of the DOT-Unit test harness.

From these initial results, however, DOT-Unit appears to scale well relative to database size, and the execution times are of the same order of magnitude as those resulting from DBUnit. This suggests that the intensional approach may provide a good compromise between saving expensive programmer time in developing new test cases and expenditure of cheaper processing time in executing the test cases.

5.2 Effect of Constraint Complexity

A further concern was the effect of increasing constraint complexity on the performance of DOT-Unit test cases. How much additional overhead is added for conditions involving a higher number of selection conditions and (most importantly) joins? In order to assess this, we grouped the test cases into three groups, according to their complexity:

- `A`: queries with one or more selections and no joins,
- `B`: queries with one or more selections and a join between two relations,
- `C`: queries with one or more selections and joins between three relations.

This gave a test suite with 5 test cases in each of these categories, which we executed against a randomly generated database state with 500 tuples per relation that does not satisfy any of the test case pre-conditions. Figure 8 shows the results obtained for the three complexity categories. We measured the average time taken to execute the test cases in each category, including a breakdown of where the time is spent in each case:

- `Test`: the time required to execute the procedural aspects of the test case;
- `Query`: the time required to execute the query aspect of the test case condition;
- `Prepare` the time required to execute the preparation aspect of the test case condition.

While the overall time required to execute the test cases rises as the complexity rises (unsurprisingly), the relative proportions of time spent in the various phases remains roughly the same. The preparation phase seems to account for slightly more than half of the time in each case, indicating that significant improvements could be achieved with a less-naive preparation algorithm.

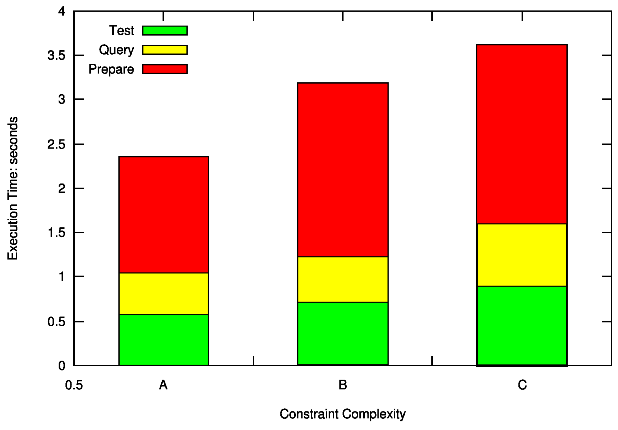

Figure 8: The Affect of Changing Constraint Complexity

6. CONCLUSIONS

We have presented a new approach to the specification of test cases for database systems that attempts to reduce the amount of manual intervention required in between test case runs while also minimising the number of spurious test failures due to inappropriate input database states. The approach has the further advantage that it sits naturally on top of test data sets taken from live databases, and this allows testing to be carried out using realistic data sets without requiring significant programmer effort to tailor the data set to the test cases. In effect, the intensional approach we have described allows software developers to trade programmer time for test execution time

Our experience has indicated that intensional test cases are quick and natural to write for anyone who is familiar with SQL and database programming, although a study with an independent testing team would be necessary before we can make any strong claims in this regard. However, compared with what is involved in writing pure JDBC database test cases and DBUnit test cases, we found that the self-contained nature of the intensional test cases was a definite advantage. Writing DBUnit test cases requires the programmer to continually check that the test case is compatible with the database description. Moreover, since it is common to try to reuse database descriptions for multiple test cases by combining their requirements into one database state, it becomes very easy to break one test case by changing the database description in order to ready it for another. These problems do not arise with intensional testing, since all the information about the test case is present in a single file (the Java class file).

We designed this first version of the preparation algorithm for simplicity and correctness rather than efficiency, and as such it performs rather stupidly in many cases. We are currently exploring options for improving the algorithm, including more intelligent selection of the order in which the relational algebra tree is traversed, alternating between passing query bindings and passing literal value bindings as is most efficient, and making use of modifications to existing tuples as well as simply adding and deleting tuples (both of which are comparatively expensive operations). The complexity of the conditions we can handle is at present limited by the capabilities of DBMonster, and can be expanded by development of a custom data generation facility. We also need to expand the range of queries that can be handled, beyond simple select-project-join queries. For example, standard SQL also allows aggregation and ordering within queries—both of which offer challenges in terms of automatic preparation.

A further problem with our current algorithm is that it may sometimes fail to find a solution to the database preparation problem, even though one exists. This is due to the fact that updates are made at leaf nodes before the full set of constraints on those nodes has been encountered. It should be possible to address the problem with more sophisticated querying techniques (this is an example of a fairly standard constrained search problem, after all), although this will add to the performance overhead. A thorough study of the trade-offs between spurious failures and more intelligent searching will need to be carried out before any concrete recommendations can be made.

Finally, we note that where it is important to test large numbers of frame constraints (i.e. aspects of the original database state that are not affected by the execution of the program under test), it may be easier to express the test case using DBUnit, rather than cluttering up the intensional test with many such constraints.

Our work presents a number of possible avenues for future work beyond the improvements mentioned above, of which the most urgent is the question of ordering of test cases within suites. This ordering can be in terms of reducing the cost of the modifications to database state or to maximise fault coverage. There is also the question of whether the modifications to database state should *always* persist between test cases or under certain conditions discarded. For example, a test case may specify that a relation be empty and to satisfy the condition the content is discarded. However, this relation may be required by later test cases and so by discarding its contents we increase the divide between the test state and the real world. This could be accomplished by either embedding the modifications inside of a transaction which can then be aborted or by using a hypothetical database engine.

7. ACKNOWLEDGMENTS

We thank Leonardo Mariani and the anonymous reviewers for comments on earlier drafts of this paper. David Willmor is supported by a research studentship from the UK Engineering and Physical Sciences Research Council.

8. REFERENCES

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 68–79. ACM Press, 1999.

[2] L. E. Bertossi and J. Chomicki. Query answering in inconsistent databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*, pages 43–83. Springer, 2003.

[3] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the SIGMOD Conference*, pages 143–154. ACM, 2005.

[4] L. Bravo and L. E. Bertossi. Logic programs for consistently querying data integration systems. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10–15. Morgan Kaufmann, August 2003.

[5] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 260–271. ACM, June 2003.

[6] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–157, August 2000.

[7] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 14(1):17–44, 2004.

[8] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. In L. E. Bertossi, A. Hunter, and T. Schaub, editors, *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 119–150. Springer, 2005.

[9] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

[10] S. M. Embury, S. M. Brandt, J. S. Robinson, I. Sutherland, F. A. Bisby, W. A. Gray, A. C. Jones, and R. J. White. Adapting integrity enforcement techniques for data reconciliation. *Information Systems*, 26(8):657–689, 2001.

[11] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1389–1408, 2003.

[12] A. Guessoum and J. W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.

[13] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 95–106. Online Proceedings, January 2005.

[14] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 98–107. ACM, September 2003.

[15] R. Langerak. View updates in relational databases with an independent scheme. *ACM Transactions on Database Systems (TODS)*, 15(1):40–66, 1990.

[16] P. Louridas. Junit: Unit testing and coding in tandem. *IEEE Software*, 22(4):12 – 15, July-Aug 2005.

[17] J. Melton and A. R. Simon. *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[18] H. Shu. Using constraint satisfaction for view update. *Journal of Intelligent Information Systems*, 15(2):147–173, 2000.

[19] D. Willmor and S. M. Embury. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Workshop (UKTest)*, pages 123–133. The University of Sheffield, September 2005.

[20] D. Willmor and S. M. Embury. A safe regression test selection technique for database–driven applications. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 421–430. IEEE Computer Society, September 2005.