Exploring test adequacy for database systems

David Willmor and Suzanne M Embury

Informatics Process Group, School of Computer Science,
University of Manchester, Oxford Road, Manchester, M13 9PL, United Kingdom
d.willmor|s.m.embury@cs.man.ac.uk

Abstract

Database systems are an important asset for many businesses. As such, it is important to test database systems thoroughly, as any faults that remain hidden may significantly impact critical business processes. However, these systems bring additional complexities that make them amongst the most complex and difficult kind of system to test. While software testing in general is a well-developed area, techniques specifically aimed at testing database systems are still in their infancy. In this paper, we present a family of test adequacy criteria for database systems that can be used to determine the "quality" of a test suite. These criteria consider various aspects of database systems including the source code structure (in terms of patterns of database operations), the existence of define–use pairs between database operations and the interactions between different applications of the database system. The criteria we present differ from existing adequacy criteria as we focus on a general definition of a database test case that is based on intensional constraints. This overcomes the problems associated with adequacy being constrained to a single static database state. We also consider transactional operators that alter the behaviour of a database system and influence adequacy.

Keywords: *software testing; database systems; test adequacy criteria*

1. Introduction

Database systems are an important asset for many organisations, since they contain vital business data (both historical and current) and support critical business processes. Because of this the testing of database systems is an important concern. Database systems often present a strong integration of computation, data and communication aspects. Not only do they incorporate a secondary persistent *database state* that is used for the storage and querying of data, but they are also often spread across a number of logical tiers. The type of system typically considered by research into software testing exists solely within a single program state, which is volatile in nature, and so their testing reflects this. Techniques are often focussed on: the definition and use of variables; the passing of control; the coverage of statements and paths; and outputs from the system. Each of these techniques is based on the computational aspects of the software system — how an output comes to exist. However, when testing database systems, data and communication aspects must also be explicitly considered during testing. Such considerations result in new testing challenges.

A database system consists of two forms of state: program and database. These states are not simple variants of each other; they are fundamentally different in several ways. Where program state is organised as a collection of named locations where individual data values can be stored, database state is organised relative to the concepts provided by a data model (such as the relational model). Large segments of a database state can be accessed or modified through the execution of a single program statement in contrast to a typical program statement which will typically define at most one variable. Moreover, changes to the state are controlled by a transaction mechanism and may be undone by some later statement. Understanding how the state may be changed is further complicated due to the huge space of possible database states. Finally, database state is persistent, so that changes made during one execution may affect the behaviour of later executions. Whilst one test case may execute correctly, its effect on the database state may affect other test cases, possibly causing an error. This can result in errors that are hard to isolate and correct.

The communication aspects of a database system handle the movement of data between program and database state. Program and database states communicate using database operations often in the form of SQL, a powerful set–oriented language with explicit semantics. SQL is used for both the definition of the database schema and for querying the data captured by it. Often, some form of database access layer (such as JDBC for Java) is used to allow SQL queries to be embedded directly into the program code. The location of embedded queries (where communication occurs between the program and the database) are likely locations of faults. Therefore, these locations often are a focus of the software testing process for database systems.

Often, in a business setting, multiple programs will interact with a single database. For example, a sales application will modify the stock levels of items to reflect purchases, a warehouse application will also modify stock levels to reflect deliveries and a management system may analyse stock levels whilst investigating trends. Due to the persistent nature of database state, the effect that one system has on the database will affect the behaviour of other completely separate systems. Whilst this effect may often be intentional it may lead to unanticipated behaviours and possible faults.

In this paper, we present a generalised view of what a database system and test case is that can be applied to the many different types of database systems that exist (Section 2). This view is based on the concept of intensional constraints that allow real world (and dynamic) database states to be used for testing. In Section 3, we present a family of test adequacy criteria based on structural and data–oriented aspects of a database system. These criteria allow us to address what sufficient testing is. These criteria also differ from those presented in existing work as they are based on the reasoning of a dynamic database state. Thus, our adequacy criteria are not constrained to one specific instance of database state. Finally, in Section 4 we present concluding remarks and discuss possible avenues for future work.

2. A view of database testing

Considering the widespread use of database systems there has been relatively little research into their testing. The work that has been produced differs by a number of factors, not least in the terminology that is used. In order to provide consistency in this paper we use the following terminology:

Application : a software program designed to fulfil some specific requirement. For example, we might have separate application programs to handle the entry of a new customer into the database, and to cancel dormant accounts once a time-limit has passed.

Database : a collection of interrelated data, structured according to a schema, that serves one or more applications.

Database application : an application that accesses one or more databases. A database application will operate on both program and database state.

Database system : a logical collection of databases and associated (database) applications.

Testing is more difficult (or, at least, different) when dealing with database applications. The full behaviour of a database application program is described in terms of the manipulation of two very different kinds of state: the program state and the database state. It is not enough to search for faults in program state, we must also generate tests that seek for faults that manifest themselves in the database state *and* in the interaction between the two forms of state. A further complication for testing is that the effects of changes to the database state may persist beyond the execution of the program that makes them, and may thus affect the behaviour of other programs [18]. Thus, it is not possible to test database programs in isolation, as is done traditionally in testing research. For example, a fault may be inserted into the database by one program but then propagate to the output of a completely different program. Hence, we must create sequences of tests that search for faults in the interactions between programs. This issue has not yet been considered by the testing research community. This has been shown to be particularly important for regression testing where the change to the functionality of one program may adversely effect other programs via the database state [18].

The literature on testing database systems varies in a number of ways. A fundamental difference in the literature is in the understanding as to exactly what a database system is. Each definition is constrained to a particular situation. There is no definition general enough to be applied to the different scenarios in which database systems may be used. The simplest view is when a single application interacts with a single database [3, 4, 8, 7]. This has been moderately extended to handle the situation in which multiple databases exist [13]. Whilst the situation in which multiple applications interact with a database has been considered in a constrained form [12, 18] there does not exist a generalised definition that is applicable to both this situation and the previous ones. Therefore, the following is a general definition of a database system that is applicable to all existing work on database testing:

Definition 1. *A database system consists of:*

- *a collection of database applications $P_1, P_2, \ldots, P_n$,*
- *a collection of databases $D_1, D_2, \ldots, D_m$,*
- *a schema Σ describing the databases.*

Conceptually we can view each individual database as a single logical database D that matches the data model Σ. Multiple databases are often used as from an implementation perspective they are easier to understand, manage and optimise. Also, database systems are often not constructed from scratch they often must use existing databases. We do not constrain Σ to a particular data model, for example relational [5, 6], object–relational [6, 17], object–oriented [1, 6, 14] etc..., however for the remainder of this paper for readability we assume that it is relational.

As with the definition of a database system there is no agreed view as to what a database test is, but an informal consensus is beginning to emerge. The following is a definition of database test cases and suites that can form the foundation for the proposals for test adequacy criteria (described in the next section) and for future work. A test case usually involves stimulating the system using some form of input, action or event. The output from the system is then compared against a specification describing what is expected and any faulty behaviour identified. In terms of database systems, the concept of a test case becomes more complicated. Not only must we consider program inputs and outputs we must also consider the input and output database states. A database test case must therefore describe what these database states are. For initial database states, existing proposals either adopt an extensional approach [13] or do not consider database state on a per test basis instead specifying a fixed initial database state for all tests [3, 4, 7, 8]. For output states, existing approaches adopt either an extensional approach [13] or intensional approach [3, 4, 7, 8].

A robust approach for testing database systems should specify both initial and output database states intensionally. This allows test cases to be executed on a variety of different states (often real world or changing states) allowing for more realistic testing. Before justifying this we present our definition of a database test case and then discuss the advantages of an intensional approach:

Definition 2. *A test case t is a quintuple $\langle i, \Delta_c^i, P, o, \Delta_c^o \rangle$ where:*

- *P the program on which the test case is executed,*
- *i is the application input,*
- *Δ_{ic} are the intensional constraints the initial database state must satisfy,*
- *o is the application output, and*
- *Δ_{oc} are the intensional constraints the output database state must satisfy.*

In this definition P, i and o represent the same concepts as the traditional notion of a test case. The database aspects of the test case are described by constraints Δ_{ic} and Δ_{oc}. We have chosen to specify the input and output database states using intensional constraints as they allow us to address a number of limitations with extensional states. In terms of input states, extensional states are: difficult to store, especially where either database states or test suites are large; difficult to maintain as each state must often be modified to reflect changes to the test case, application or data model; and difficult to ensure they reflect the real–world and changes to the database state that may occur over time. In terms of the output state, extensional states are: expensive to determine if two large states are identical; difficult to maintain as the output state must be modified to reflect changes to the input state and the functionality of the system; and time consuming to manually create states that reflect complex behaviour that a test case may exhibit on the initial state. Our intensional technique specifies constraints that a test case must satisfy to determine (a) applicability (if the input state is valid for the test case) and (b) success (if the output state is correct). Consider the following very simple example in which a new customer is added to the database:

`Test Case 1:  add a new customer with <name>, <email> and <postcode>`

- Δ_{ic} `:  initial state constraint`
  1. `no customer C in CUSTOMER has C.NAME=<name>, C.EMAIL=<email>`
     `and C.POSTCODE=<postcode>`
- Δ_{oc} `:  output state constraint`

1. `at least one customer C in CUSTOMER has C.NAME= <name>, C.EMAIL=<email> and C.POSTCODE=<postcode>`

This test case is relatively simple and imposes a single input constraint that specifies that no customer should exist in the database that matches the customer to be added. The output constraint specifies that after executing the test case the database should contain exactly one customer matching the customer to be added. We specify exactly one customer in the output constraint as it allows us to cover faults where no customer was added and where multiple customers were added. The use of intensional constraints against a real–world database raised the question of how we can deal with situations in which the initial constraint does not hold. This is important as whilst using a real–world database state provides us with realistic data, we cannot create opportunities for exposing faults that might arise in the future, but which are not present in existing data.

A test case aims to test a particular use of a system. However, database systems exhibit significantly more complex functionality. For example, a sequence of related tasks may be carried out by a user interspersed with tasks of other users. Tasks may also be spread across a number of individual programs. These cannot be captured by the execution of a single test case since our definition of a test case assumes a single program execution. Consider the situation in which a test case t_1 adds an item to a shopping cart and t_2 increases the quantity of the item added. If t_1 does not correctly add the item, it is not possible for t_2 to increase its quality. Therefore, the execution of t_2 may fail not as a result of a problem with the program but because t_2 is dependent upon t_1. This dependency problem can be addressed by modifying database state to satisfy the initial constraints. However, this approach has a number of limitations. The simplest are due to the resources required for generating database states. The most important is due to the fact that whilst we can satisfy t_2s requirements from t_1 we are unsure if t_1 has an unforeseen impact on t_2. For example, a test case may change part of the database state that can adversely affect the behaviour of a subsequent test case. Therefore, it is obvious that certain behaviours require the execution of individual tests in an ordered sequence. A *test sequence s* is a sequence of test cases $\langle t_1, \ldots, t_n \rangle$. Each test of the sequences is executed in the specified order. If a test case does not meet its output conditions (the test fails) the user is notified of the failure. The database state is then modified to allow the sequence to proceed. However, the test result of the sequence is flagged to tell the user that it did not execute correctly. This is done, instead of simply stopping the sequence, as the tests still provide a certain confidence in the system. Our approach to test sequences allows an individual test case to exist in a number of test sequences. It can also be observed that test sequences can be used for more than testing complex functionality. It can potentially take a lot of effort to set up a database for a particular test case. If several test cases require similar input databases, then it will be much more efficient to run them all against the same database. For example, consider the situation where a database contains records for customers. In an example sequence, the first test case would create a customer; the second would modify the customer; and the third would delete the customer. Each test case represents important functionality of the system which are all related through the use of the same customer. It is therefore more efficient to use a sequence to group related test cases.

3. Test adequacy criteria for database systems

A test suite is a collection of test cases (or test sequences in our approach) usually targeted towards the verification of the entire system or a specific section of the system. The manner in which a test suite is generated varies between different situations. The simplest is to randomly generate tests for the system. However, it is common to use more formalised techniques based on some aspect of the system, including: the systems specification, observations of the system being used, and the structure of the systems source code. To test every possible input is impractical for anything but the simplest of programs. For database systems this becomes impossible. Thus, if we cannot completely test a system, what is sufficient testing? Current work into software testing has proposed a number of test adequacy criteria that if satisfied will sufficiently test the system according to some characteristic of the specification or implementation. In terms of database testing only one set of criteria exist for determining the quality of a test suite [13]. This approach is based on determining for each database operation an extensional representation of the portion of the database defined or referenced by the application. This approach is limited for a number of reasons: (a) It generates an extensional representation in which non–determinism countered by using a fixed initial state. However, this means that the test suite can only be considered valid for this particular state. (b) It does not consider the effect transaction operators have on the definition–use pairs and so will include unnecessary test cases for interactions that will not occur. (c) Nor does it consider multiple applications or instances of the same application accessing a single database. The only other work on test adequacy for database systems is by Suárez-Cabal and Tuya [2] in which they present a metric for testing coverage of an SQL SELECT query, and a method for detecting when tuples needed to be added to the database to ensure better coverage, based on analysis of the corresponding query tree[1]. Their work aims to determine adequacy of a single query (specifically the SELECT query) and does not consider the behaviour of the system as a whole.

In this section we present a number of test adequacy criteria based on our intensional specification of database state and intensional descriptions of the behaviour of database operations. The criteria presented, are focussed on the structural and data-oriented elements of database systems. Briefly, structural elements include branches, loops and procedure calls. Data-oriented elements are the points at which data is defined and used in the program. However, first we present a brief discussion about the types of faults that can occur within a database system.

The fundamental issue in database testing is whether the application behaves as specified [3, 4]. From a simple perspective this can be seen as determining if the output from a database system matches its required output. Bearing in mind that database state is persistent, it can be observed that the output database state is dependent not only on the input to the program but also the initial (or input) database state. Therefore, a test case execution can be seen as moving the database from one state to another. It is this transition that we aim to verify. The first type of fault is simply that the implemented functionality does not match the specified functionality. Other faults include: attempting to access a database entity that does not exist, operations attempting to violate the databases constraints (such as primary key or referential integrity), and transactions being aborted or committed incorrectly. These types of faults manifest themselves either in the database state or as a result of an interaction between program and database state.

3.1. Structural test adequacy criteria

Structural test criteria are a commonly used software test adequacy criteria [20]. This form of testing is based on a structural model that represents the physical implementation of the software application. Kapfhammer and Soffa's [13] approach to test adequacy is based on an extended version of a control flow graph in which extra edges are included to capture the dependencies that exist between database operations. However, this model only captures dependencies that exist between database operations in a single procedure. Instead we base our model on a representation that completely describes the structure of a database system and its composite components.

Definition 3. *Each application P of the database system is modelled as an interprocedural graph consisting of:*

- *C_P, the set of control flow graphs, where each $c_i \in C_P$ corresponds to a procedure m_i of program P.*
- *I is a graph where each edge represents a procedure call.*

This is a general model that can be tailored towards particular implementations. For example, for a Java program I is an interclass relation graph (IRG) representing the relationships between the classes (and their methods) in P [10]. The IRG models the complexities associated with object–oriented languages, including variable and object type information; internal and external methods; interprocedural interactions; inheritance, polymorphism and dynamic binding; and exception handling. In the model described in Definition 3 each statement of a program is captured by a particular node. These nodes can be categorised and annotated with additional information. In particular we use the concept of a database operation type:

Definition 4. *A database operation δ is a node of a control flow graph that consists of some form of interaction with a database D. Each δ is an notated with;*

- *$\delta.\Sigma_{add}$ the subset of D that is updated by δ,*
- *$\delta.\Sigma_{del}$ the subset of D that is deleted by δ, and*
- *$\delta.\Sigma_{read}$ the subset of D that is read by δ.*

The sets $\delta.\Sigma_{add}$, $\delta.\Sigma_{del}$ and $\delta.\Sigma_{read}$ allow us to reason over the interactions with the database and possible relations between different database operations. The subsets of the database are represented intensionally as relational algebra expressions.[2]

[1]A query tree is conceptually similar to an abstract syntax tree for programming languages.
[2]For more information about how these sets are generated , reasoned over and can be used please see [18, 19].

Our structural test adequacy criteria are based upon a static analysis of the model according to a specific criterion. These structural criteria can be seen as analogous to traditional control flow–based criteria but with the additional characteristics of database systems incorporated. Similar to control–flow based techniques we utilise the concept of a complete path π that starts at the graph's entry node and ends at an exit node [9]. Intuitively, it can be observed that the execution of a particular test case will result in the execution of a particular complete path (relative to the input to the program and the state of the database at that moment in time). Therefore we use the notation π_t to refer to the complete path relative to the execution of test case t.[3] As our model is based upon control–flow graphs, existing criteria, such as statement, branch and path coverage, are applicable to database systems. For brevity we refer the reader to [20] for a detailed description of these criteria. The use of complete paths in test adequacy criteria is complicated by the fact that we describe initial states using intensional constraints. Therefore, multiple executions of a test case may result in different paths due to changes in the database state. A test suite is therefore only adequate in terms of the state in which it was executed. For example, a test suite may only satisfy a subset of the requirements of a criterion when executed against a particular state, however, as that state changes so may the degree of satisfaction. The test suite executed at a later date may result in a different degree of coverage.

The database system model includes a special type of node for each database operation in an application. Criterion 1 simply assesses coverage of all database operations without discriminating as to their effect. Whilst a fault may not be caused directly by a database operation, it is through these statements that database-related faults become detectable, either by propagation to the database state through state change or by retrieval from the database state. Since each database operation may potentially reveal a fault, ensuring coverage of all such statements by a test suite gives some guarantee that a wide variety of database faults will be detected.

Criterion 1. *A test suite ts satisfies the **All Database Operations** criterion if for each database operation δ, there exists a $t \in ts$ such that $\delta \in \pi_t$.*

Criteria 2 and 3 assess coverage relative to two of the most common types of interactions with a database: retrieval and update. They each potentially require fewer test cases than the 1 criterion, and so can be more efficient when dealing with kinds of programs where there is some expectation as to the kind of fault that may appear. For example, when testing a set of batch applications, we may prefer to concentrate our testing effort on updates to the database, ensuring that the result states produced by the programs are correct. In such programs, database retrieval is used only to pull data into memory in order to manipulate it before writing it back to the database. Any errors in this part of the code will likely result in incorrect update operations as well. Alternatively, where a system consists of many report-style or query-browsing applications, we may wish to focus attention on how data is retrieved from the database, and how it is later manipulated before being presented to the user, rather than the minor house-keeping updates that occur during report generation.

Criterion 2. *A test suite ts satisfies the **All Read Operations** criterion if for each database operation δ where $\delta_{read} \neq \emptyset$, there exists a $t \in ts$ such that $\delta \in \pi_t$.*

Criterion 3. *A test suite ts satisfies the **All Write Operations** criterion if for each database operation δ where $(\delta_{add} \neq \emptyset) \vee (\delta_{del} \neq \emptyset)$, there exists a $t \in ts$ such that $\delta \in \pi_t$.*

Although faults may be visible to a particular program in the middle of a transaction, the key point at which they become fully visible externally is when the changes made by the program are made durable by execution of a transaction commit. These are key points in the program, when a set of related changes is declared to be either consistent (i.e. legal) and can therefore be made durable within the database state, or inconsistent and must therefore be undone. Therefore, by ensuring that all such operations are executed at least once by a test suite, we have some guarantee that our test suite is exercising a significant proportion of the kinds of database interactions that the application programs implement. This leads us to propose the criterion 4 in which all commit and abort statements are required to be exercised by the test suite for it to be considered adequate. In general, this criterion will allow smaller test suites to be used than criteria 2 and 3.

Criterion 4. *A test suite ts satisfies the **All Commits and Aborts** criterion if for each database operation δ where $type(\delta, commit) \vee type(\delta, abort)$, there $\exists\, t \in ts$ such that $\delta \in \pi_t$.*

The above adequacy criteria are all subsumed by the standard *all–statements* criterion,[4] and therefore share its inherent weaknesses, in that test suites may satisfy them but may still leave a large part of the control flow graphs of a set of

[3] This is often referred to as the execution trace of t in the literature.

[4] Sometimes referred to as all–nodes.

programs unexplored. In our context, this disadvantage is most serious in the case of the weakest criterion (4). The motivation behind this criterion is that the test case should execute all transactions within the programs. However, the structure of most database programs is much more complex than this criterion implies. There will in general be many ways of reaching a specific commit or abort operation. Many of these will be slight variants on the same basic transaction behaviour, but others may represent very different transactions that need to be tested.

In other words, rather than being satisfied with testing one path to each commit or abort, we would ideally prefer to test all such paths. In order to define such an adequacy criterion, we first require a notion of a *transaction path*.

Definition 5. *A transaction path in a program P is a subpath $n_i, \ldots, n_j$ in a complete path of P where:*

- *the node immediately preceding n_i is either* START *or a commit or abort operation,*
- *n_j is either a commit or an abort operation, and*
- *the subpath $n_i, \ldots, n_{j-1}$ is commit- and abort-free.*

Based on this, we can now define an adequacy criteria that requires all transaction paths to be exercised by a test suite:

Criterion 5. *A test suite ts satisfies the **All Transactions** criterion iff every transaction path in P is covered by some test $t \in ts$.*

In practice, of course, such a criterion would need to be used in conjunction with some mechanism for ensuring that the presence of cyclic paths does not lead to an infinite number of transaction paths to be tested. For example, Howden's boundary interior criterion could be applied [11]. If testing resources are limited (as is usually the case) we may also want to avoid repeated testing of very similar transaction paths, and would instead want to concentrate testing effort on covering as wide a variety of transaction paths as possible. This would require some heuristic to be combined with the criterion, in order to determine which transaction paths are deemed sufficiently different from those already explored to be worth attention.

In database systems, we have an additional source of structure that can form the basis for further test adequacy criteria. This is the structure of the database itself. A common strategy when testing database applications, for example, is to choose tests that cover all parts of the schema,[5] and all forms of operation on each schema element. For example, if a new database system is created that includes a *Customer* table, we would expect there to be code that controls the addition of new customers to the database, that handles modifications to their details (such as name or address) and that carries out deletion of data for customers that are no longer deemed to be active. We would also expect there to be at least one program that reads data from the *Customer* table, since if it is not used then there is little point in maintaining the data.

A well designed database system will often be created with sub-routines or sub-programs that handle these basic updates, and which ensure that the same business logic is applied, regardless of what higher level application program they are called from. (Such systems are often constructed as three-tier systems, with an upper interface layer, a middle business logic layer and a supporting database services layer.) In testing such systems, we may wish to ensure that each form of operation on each part of the schema has been tested at least once, rather than testing many calls to the same operations. This leads to a further structural adequacy criterion, which we define here in terms of the relational data model (though the same principle could easily be applied to other models):

Criterion 6. *A test suite ts for a database system with schema Σ satisfies the **All Schema Elements** criterion iff for every relation $r \in \Sigma$, the following operations are covered by at least one test case in ts (though not necessarily the same test case):*

- *an operation which retrieves data from r,*
- *an operation which inserts new tuples into r,*
- *an operation which deletes tuples from r, and*
- *an operation which modifies some existing tuples in r.*

Further variants of this criteria can be considered, which operate a finer granularity of database structure. For example, we might wish to ensure that our test case will include operations which read from each attribute in each table, as well as

[5] Or, since databases are often required to support a wide variety of applications, coverage may be limited to those parts of the schema actually accessed by the programs to be tested, which can (in general) be determined statically.

modifying the attribute. Another common area of focus for testing in database applications is on the relationships between tables (modelled using foreign keys and additional integrity constraints in relational systems), since errors in modelling the cardinality and optionality of relationships are common in database programming, and can have severe ramifications.

3.2. Define–use test adequacy criteria

Traditional programs are based around the definition and use of variables [15, 16]. A *definition* occurs when a variable is on the left hand side of an assignment. A use may either occur: (a) on the right hand side of an assignment (a computation–use) or (b) in the predicate of a conditional logic statement (a predicate–use). A *definition–use pair* occurs between a statement that defines a variable and a subsequent (in the control flow graph) statement that uses that variable and there is no intervening definition. A number of different criteria have been proposed based on the concept of definition–use pairs [15, 16]. The simplest include the all-uses criterion (in which all uses must be covered) and the all-du-paths criterion (in which all paths between definition–use pairs must be covered) [15, 16].

For database applications define–use pairs relate to the definition and use of parts of the database. Whilst, this is more complicated than program statements, it has been successfully employed in testing [13, 18] and program slicing [19]. We will utilise the approach we proposed previously in the context of slicing [19] and regression testing [18] as it has a finer level of granularity than the approach of Kapfhammer and Soffa [13] and also includes the effects of the transaction operators: commit and abort. This is important as a definition–use pair cannot exist if the definition has been aborted before it is used. The following is a description of a database definition–use pair:[6]

Definition 6. *A database definition–use pair exists between operation δ_1 and operation δ_2 iff:*

1. $\delta_2.\Sigma_{read} \cap (\delta_1.\Sigma_{add} \cup \delta_1.\Sigma_{del}) \neq \emptyset$*, and*
2. *there is a rollback-free execution path p between δ_1 and δ_2 such that:*
   - $\delta_1.\Sigma_{add} \setminus p.\Sigma_{del} \neq \emptyset$ *o r*
   - $\delta_1.\Sigma_{del} \setminus p.\Sigma_{add} \neq \emptyset$

For brevity, we use the notation $\delta_1 \dashrightarrow \delta_2$ to denote the fact that there exists a definition–use pair between δ_1 and δ_2. In the above description it can be observed that definition–use pairs will only be encountered if certain paths of the program are traversed (some paths will not be rollback–free). Therefore, in order to specify coverage we specify that $\Pi_{\delta_1 \dashrightarrow \delta_2}$ is the set of all paths in which the specific definition–use pair will occur. Using this description of a database definition–use pair it is possible to propose a criterion based on their coverage:

Criterion 7. *A test suite ts satisfies the **All Database Application Define–Use Pairs** criterion if for each $\delta_1 \dashrightarrow \delta_2$, there exists a $t \in ts$ such that $\pi_t \in \Pi_{\delta_1 \dashrightarrow \delta_2}$.*

In this criterion each instance of a define–use pair should be matched by a test case in which the use occurs after the definition in the path of the test case. Intuitively, it can be observed that the above definition–use pairs only exist within a single program. However, given the persistent nature of database state data is shared between different programs. Therefore, a program may define a part of the database that may be subsequently used by another program. Given this situation we are able to specify a definition–use criterion across the entire database system:

Criterion 8. *A test suite ts satisfies the **All Database System Define–Use Pairs** criterion if for each $\delta_1 \dashrightarrow \delta_2$ where $\delta_1 \in P_1$, $\delta_2 \in P_2$, there exists a test sequence $s \in ts$ where the complete path of the test sequence π_s contains a rollback–free subpath p between δ_1 and δ_2 such that:*

- $\delta_1.\Sigma_{add} \setminus p.\Sigma_{del} \neq \emptyset$ *o r*
- $\delta_1.\Sigma_{del} \setminus p.\Sigma_{add} \neq \emptyset$

This criterion aims to cover each instance of a define–use pair that may exist within an entire database system. For each define–use pair a test sequence should exist in the test suite in which the definition occurs and then subsequently used. The rollback-free subpath condition checks that between the definition occuring and it being used its effect on the database state has not been reversed (either through an abort command or an intervening definition).

[6]Please refer to [19] for details describing how we reason over database queries and construct the definition–use pairs (which are described as database–database dependencies).

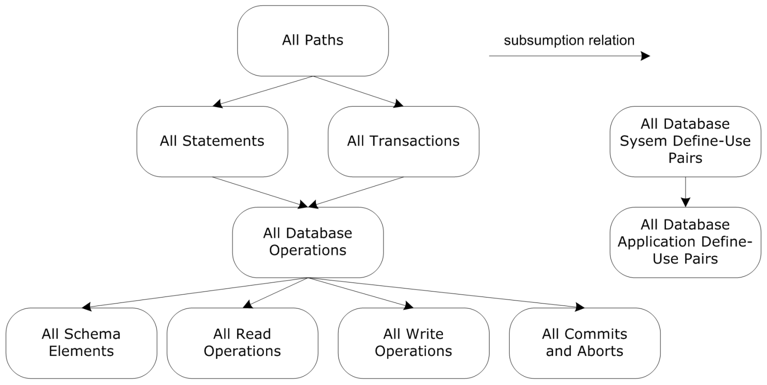

Figure 1: Test adequacy criteria subsumption hierarchy

3.3. Subsumption hierarchy of test adequacy criteria

The test adequacy criteria proposed in this paper do not produce mutually exclusive test suites. We discuss the inclusiveness of two test adequacy criteria in terms of subsumption. A criterion C_1 *subsumes* a criterion C_2 if every test suite that satisfies C_1 also satisfies C_2. Figure 1 shows the subsumption hierarchy for the criteria proposed in this paper. The relationships presented in this hierarchy are justified as follows:

- *All Database Operations subsumes All Read/Write/Commits&Aborts Operations and All Schema Elements*
  The simplest of subsumption relationships. Each of type read, write, and commits & aborts are a type of database operation. Therefore, a test suite that covers all database operations will inherently cover each of the types.
- *All Statements subsumes All Database Operations*
  A database operation is a particular type of statement that interacts with the database. Therefore, a test suite that covers all statements inherently covers all database operations.
- *All Transactions subsumes All Database Operations*
  A transaction is a collection of database operations. Every database operation must exist in a transaction.[7] Therefore, a test suite that covers all statements inherently covers all database operations.
- *All Paths subsumes All Transactions*
  A transaction is captured by a transaction path. This in turn is a subpath of the systems source code. Therefore, to cover all paths (which inherently covers all subpaths) will cover all transactions.
- *All Database System Define–Use Pairs subsumes All Database Application Define–Use Pairs*
  Define–use pairs of the system as a whole will also include all define–use pairs located in an individual program. Therefore, to cover all database system define–use pairs inherently covers all database application define–use pairs.

4. Conclusions and future work

In this paper, we have addressed a number of fundamental issues regarding database testing, particularly: (a) what a database system is? (b) what a database test case is? and (c) what is adequate testing of a database system? In response to this, we have presented the following contributions in this paper:

- Basic definitions for database testing:

[7]A number of database programming languages (and access layers) allow certain operations to be performed outside of transactions or operate in an auto commit mode (in which each operation is automatically committed if it is successful). We therefore view each operation outside of a transaction as existing within its own individual transaction.

  - A database system that is applicable to both the types of systems in use with businesses and the currently existing techniques on database testing. Our definition is based on the concept that a database system may consist of one or more applications that interact with one or more databases.
  - A database test case. This definition uses intensional constraints to specify requirements required of the initial and output database states. The use of intensional constraints (particularly for the input state) allows us to utilise both real–world and artificial database states for testing.
  - A database test sequence. This definition describes the execution of a sequence of database test cases aimed at verifying some form of complex behaviour. Often the behaviour of a system cannot be verified by a single test case. A test sequence also allows us to group test cases that can operate on the same initial state without affecting each other.

- Test adequacy criteria for database systems that aim to determine what "adequate" testing is:

  - *Structural* criteria focus upon the structural aspects of the database system. We focus on two forms of structural information: the application source code and the data model. The source code based criteria are based on the possible patterns of database operations that may be executed. Particularly, we have presented criteria based on covering different types of database operations, transactional statements and complete transactions. The data model based criterion aims to cover the different entities of the data model. This translates to covering: tables, columns, foreign key relationships, etc. . . in the relational model.
  - *Define–use* criteria focus upon the relationships between database operations in terms of what subset of the database they define or use. Our approach differs from existing techniques as the reasoning over the effect of a database operation on the database is based on intensional descriptions. This allows us to determine all of the possible define–use pairs and not just those valid for the current database state. We specify define–use criteria for a single application and the database system as a whole.

- Subsumption hierarchy describing the relationships between our test adequacy criteria and by placing them into perspective with classical program state based criteria.

This work has presented a number of avenues for further work. The first avenue is an empirical comparison between the proposed test adequacy criteria. Whilst our subsumption hierarchy provides a descriptive comparison of the differences between the criteria it does not tell us anything about the costs associated with determining adequacy, the fault coverage of different adequate test suites or the cost of executing each test suite. Furthermore, we plan to investigate possible optimisations to improve testing. This is particularly important for concurrent systems as the number of possible combination is a combinatorially explosive problem.

References

[1] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim. Data model issues for object-oriented applications. *ACM Trans. Inf. Syst.*, 5(1):3–26, 1987.

[2] M. J. S. Cabal and J. Tuya. Using an SQL coverage measurement for testing database applications. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE)*, pages 253–262, October-November 2004.

[3] 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.

[4] 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.

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

[6] T. Connolly and C. Begg. *Database Systems*. Addison-Wesley, 3 edition, 2002.

[7] Y. Deng and D. Chays. Testing Database Transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2005.

[8] Y. Deng, P. G. Frankl, and Z. Chen. Testing database transaction concurrency. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 184–195. IEEE Computer Society, October 2003.

[9] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[10] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 312–326. ACM, October 2001.

[11] W. E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, 24(5):554–560, 1975.

[12] G.-H. Hwang, S.-J. Chang, and H.-D. Chu. Technology for testing nondeterministic client/server database applications. *IEEE Transactions on Software Engineering*, 30(1):59–77, 2004.

[13] 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.

[14] C. Lécluse, P. Richard, and F. Vélez. O2, an object-oriented data model. In H. Boral and P.-Å. Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 424–433. ACM Press, 1988.

[15] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the 6th International Conference on Software Engineering (ICSE)*, pages 272–278, September 1982.

[16] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[17] M. Stonebraker, P. Brown, and D. Moore. *Object-Relational DBMSs*. Morgan Kaufmann, 2 edition, 1998.

[18] 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.

[19] D. Willmor, S. M. Embury, and J. Shao. Program slicing in the presence of a database state. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 448–452. IEEE Computer Society, September 2004.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.