A Family of Test Adequacy Criteria for Database-Driven Applications

Gregory M. Kapfhammer
Department of Computer Science
University of Pittsburgh
gkapfham@cs.pitt.edu

Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
soffa@cs.pitt.edu

ABSTRACT

Although a software application always executes within a particular environment, current testing methods have largely ignored these environmental factors. Many applications execute in an environment that contains a database. In this paper, we propose a family of test adequacy criteria that can be used to assess the quality of test suites for database-driven applications. Our test adequacy criteria use dataflow information that is associated with the entities in a relational database. Furthermore, we develop a unique representation of a database-driven application that facilitates the enumeration of database interaction associations. These associations can reflect an application's definition and use of database entities at multiple levels of granularity. The usage of a tool to calculate intraprocedural database interaction associations for two case study applications indicates that our adequacy criteria can be computed with an acceptable time and space overhead.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*

General Terms

Experimentation, Languages, Verification

Keywords

database-driven applications, test adequacy criteria

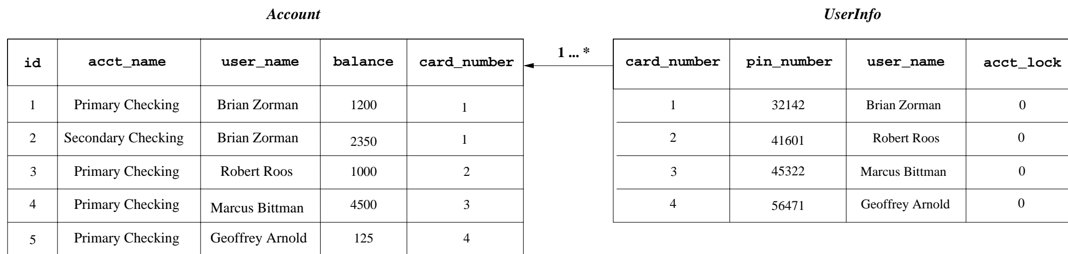
1. INTRODUCTION

The process of testing software is difficult because it requires the description of all the interfaces in a software system and the adequate testing of these interfaces. Yet, even simple software applications have complicated and ever-changing operating environments that increase the number of interfaces and the interface interactions that must be tested. Device drivers, operating systems, and databases are all aspects of a software system's environment that are often ignored during testing [21]. In recent years, a wide range of traditional software testing techniques have been proposed, implemented, and evaluated. However, relatively little research has specifically focused on the testing and analysis of applications that interact with databases.

A testing effort cannot ensure the production of high quality software if it does not consider the operating environment of the application under test. Yet, it is not sufficient to simply test the program and each aspect of the program's environment in isolation. In order to establish a high level of confidence in a software system with a complicated environment, it is necessary to develop test adequacy criteria that correctly capture a program's interaction with its environment. An adequacy criterion can be used to automatically generate test suites that are satisfactory with respect to the selected criterion. These "environment aware" adequacy metrics can also be used to assess the quality of tests that are manually created by software developers. Finally, test adequacy criteria can serve as stopping rules to determine when the testing of an application and its environment can cease [23].

Intuitively, a database-driven application is a program whose environment always contains one or more databases. Given the preponderance of high quality database management systems and the number of organizations that are now collecting an unprecedented amount of data, there is a clear need for software testing techniques that test an application and its interaction with a database. In this paper, we define a family of dataflow-based test adequacy criteria for database-driven applications. Specifically, we provide formal definitions for the *all-database-DUs*, *all-relation-DUs*, *all-attribute-DUs*, *all-record-DUs*, and *all-attribute-value-DUs* adequacy metrics as companions to the traditional *all-DUs* dataflow adequacy criterion. Tests that are adequate with respect to this family of criteria exercise all of the database interaction associations for all of the entities within a relational database. Moreover, we develop a representation of a database-driven application called a *database interaction control flow graph* (DICFG) in order to calculate our family of test adequacy metrics. We also present algorithms for constructing this representation of a database-driven application. Next, we provide an empirical analysis of the time and space overhead incurred by a tool that enumerates the database interaction associations required by our family of test adequacy criteria.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.
Copyright 2003 ACM 1-58113-743-5/03/0009 ...$5.00.



Account

id	acct_name	user_name	balance	card_number
1	Primary Checking	Brian Zorman	1200	1
2	Secondary Checking	Brian Zorman	2350	1
3	Primary Checking	Robert Roos	1000	2
4	Primary Checking	Marcus Bittman	4500	3
5	Primary Checking	Geoffrey Arnold	125	4

1 ... *

UserInfo

card_number	pin_number	user_name	acct_lock
1	32142	Brian Zorman	0
2	41601	Robert Roos	0
3	45322	Marcus Bittman	0
4	56471	Geoffrey Arnold	0

Figure 1: An Instance of the Relational Database Schema in the ATM Application.

The important contributions of this paper are as follows:

1. The definition of a database-driven application and the presentation of equations that describe the dataflow information within this type of application.
2. The definition of a family of test adequacy criteria for database-driven applications that is based on the traditional dataflow *all-DUs* criterion.
3. A representation for database-driven applications that describes a program's interaction with relational database entities.
4. An examination of the time and space overhead incurred during the generation of intraprocedural database interaction associations (i.e. associations within a method) for two case study applications.

2. AN EXAMPLE APPLICATION

In order to make our discussion more concrete, we describe a simple example application. In this application, the user is able to conduct an unlimited number of transactions with his/her account after submitting the appropriate PIN number. The currently supported transactions include the capability to deposit or withdraw money, transfer money from one account to another, and check the balance of an existing account. The relational schema of the *Bank* database that is used by the ATM application consists of the *UserInfo* relation and the *Account* relation. Figure 1 shows an example configuration of the relational schema. There is a foreign key constraint that requires any *card_number* that is specified in the *Account* relation to be associated with a *card_number* in the *UserInfo* relation. Also, the current specification of the relational schema states that there is a one-to-many relationship between the *card_number* attribute in *UserInfo* and the *Account* relation's *card_number* attribute.

3. TESTING CHALLENGES

A test adequacy measure that includes an understanding of the source code of the program and the state of the database(s) with which the program interacts must address a number of challenges. Even though the state space of an application that interacts with a relational database is well-structured (because it is described by a relational schema), it is also essentially infinite [4]. Also, the representation of a database-driven application must clearly indicate the coupling points between the program and the entities within the database [12]. Thus, the calculation of any test adequacy metric for database-driven applications requires a unified application representation to select defect-revealing test cases from an input space that is essentially infinite.

A test adequacy criterion for database-driven applications must ensure the existence of tests that can reveal the types of faults that are commonly found in programs that interact with databases. Accuracy, relevancy, completeness, and consistency are some of the aspects that are commonly associated with the quality of the data that is stored in a database [14, 18, 20]. In order to examine the types of defects found in database-driven applications, we use the conception of database integrity initially proposed by Motro [14]. Thus, we view the integrity of an application's database as a function of the validity and completeness of the data that is stored within the database. Intuitively, a database is in a valid state if it contains only records of information that reflect the state of the real world. Moreover, a database is complete if it contains all of the records of information that reflect the current state of the real world.

Using Motro's view of database integrity, it is clear that a database-driven application can cause the violation of database integrity in four different ways [14]. A program can violate the validity of one of its databases if (1-v) it inserts a record into a database that does not reflect the real world or (2-v) it fails to insert a record into the database when the status of the real world changes. A database-driven application can violate the completeness of one of its databases if (1-c) it deletes a record from a database when this record still reflects the real world or (2-c) the status of the real world changes and it fails to include this information as a record inside of a database.

A testing approach designed to isolate database-driven application defects that violate database validity and/or completeness in fashion (2) would require the ability to monitor the status of the real world. However, a testing technique could reveal faulty program operations that violate database validity or completeness in fashion (1). In fact, a testing approach that causes a database-driven application to change the database in an inappropriate fashion and then attempts to reference the inappropriate database entities will reveal program source locations that lead to validity and completeness violations of the first type. In this paper, we develop a family of test adequacy criteria that are tailored to isolate program defects that can cause type (1-v) and (1-c) violations.

As an example, suppose that the application described in Section 2 contains an `addUser` operation that inappropriately inserts a null value into a database record that contains a new ATM user's information. When executed, this defect would cause a type (1-v) violation. As another example, assume that the same ATM data management application

select $A_1, A_2, \ldots, A_q$
from $r_1, r_2, \ldots, r_m$
where Q

(a)

delete from r
where Q

(b)

insert into $r(A_1, A_2, \ldots, A_q)$
values$(v_1, v_2, \ldots, v_q)$

(c)

update r
set $A_l = F(A_l)$
where Q

(d)

Figure 2: General Examples of the SQL Operations.

provides a faulty `removeUser` operation to delete an ATM user's account based upon the unique ID associated with the user. If this operation contains a defect that interprets the provided ID as one of the ATM card numbers assigned to a user, it could erroneously remove multiple accounts and cause a type (1-c) violation. Both of these faults will only be revealed by a test case that executes the faulty operation and then subsequently executes some additional program operation(s) that attempt to use the erroneous data within the database. The anecdotal evidence provided by our past examples and several studies of the fundamental dimensions of data quality in databases [14, 18, 20] clearly motivate the usage of a dataflow-based test adequacy criterion.

4. TERMINOLOGY

4.1 Relational Database Model

Due to the dominance of the relational model in existing data-processing applications, we limit our discussion to database-driven applications that interact with relational databases [17]. The fundamental concept in the relational data model is a *relation*. A relational database can be viewed as a set of relations where a relation of attributes $A_1, \ldots, A_q$, with domains $M_1, \ldots, M_q$, must be a subset of $M_1 \times \ldots \times M_q$. That is, a relation is simply a set of *records*. Each record in a relation can be viewed as an ordered set of attribute values. A *relational database schema* describes the logical design of the relations inside the database and a *relational database instance* is a populated example of the schema [17].

4.2 Structured Query Language Operations

Among other functionality, the structured query language (SQL) includes a data-definition language (DDL) and a data-manipulation language (DML) that facilitate the definition of relational schemas and the manipulation, insertion, and deletion of relational data [17]. This paper will focus on the SQL DML operations of **select**, **delete**, **insert**, and **update** and also consider SQL statements that combine these commands in an appropriate fashion. Generally, our model of the structured query language contains all of the features of the SQL 1999 standard that are most relevant to the flow of data in a database-driven application.

This paper will rely upon the simple description of these SQL operations provided by Figure 2. First, it is important to note that parts (a), (b), and (d) of Figure 2 contain a reference to logical predicate Q. The format of Q is $V_s \Re V_t$ where $\Re \in \{<, \leq, >, \geq, \neq, \textbf{in}, \textbf{between}, \textbf{like}\}$ and V_s is any attribute from the set $\{A_1, A_2, \ldots, A_q\}$ (or, just $\{A_l\}$ in the case of the **update** statement). We define V_t in a fashion similar to V_s; however, the value of V_t can also be an arbitrary string, a pattern description, or the result of the execution of a **select** query. Finally, the $v_1, \ldots, v_q$ in part (c) of Figure 2 represent the specific attribute values that will be inserted into relation r. We will assume that all SQL operations will process relations specified by the relational schema. Furthermore, this paper will not focus on database-driven applications that use SQL DDL statements to change the relational schema at runtime.

4.3 Database-Driven Applications

Silberschatz et al. define a *database-management system* (DBMS) as a set of procedures that are used to access, update, and modify a collection of structured data called a *database* [17]. A database-driven application consists of a database, a database management system (DBMS), and a set of applications that interact with the database through the management system. Chan et al. propose a classification for database-driven applications that requires all such software systems to fall into one of two categories [2, 3]. In the first category, the applications that use the DBMS are constructed with the development environment that is provided by the management system. In the second category, the application(s) are written in a general purpose programming language that allows for the embedding of data manipulation language statements [3]. Due to the predominant usage of general purpose programming languages, we will focus on database-driven applications that belong to the second category. For the purposes of this paper, we define a database-driven application in Definition 1. Next, Definition 2 states our understanding of a database interaction point and Definition 3 explains the specificity of an arbitrary interaction point.

Definition 1. A *database-driven application* consists of a program P and database(s) $D_1, D_2, \ldots, D_n$ that are specified by the fixed relational schema(s) $S_1, S_2, \ldots, S_n$, respectively. Method m in program P can be viewed as a set of database interaction points, $I = \{I_1, I_2, \ldots, I_s\}$. □

Definition 2. An arbitrary *database interaction point* $I_r \in I$, in P's method m, corresponds to the execution of one of the SQL data manipulation language operations **select**, **delete**, **insert**, or **update** on relational database D_i. □

Definition 3. The *specificity* of a database interaction point $I_r \in I$ can be *static*, *dynamic*, or *partially dynamic*. A *static interaction point* completely specifies the aspects of D_i's relational schema S_i that it will manipulate. A *dynamic interaction point* requires the specification of the desired manipulations during the execution of P. A *partially dynamic interaction point* includes manipulations of database D_i that are dynamically and statically specified. □

Figure 3 provides the complete code listing for the `getAccountBalance` and the `lockAccount` methods. Each of these methods contains exactly one database interaction point. Since the `executeQuery` method used in `getAccountBalance` currently accepts a fixed `String` variable `qs`, it would be considered a static database interaction point. The usage of the `card_number` parameter inside of the `lockAccount` method makes this method's database interaction point partially dynamic.

```
1  public double getAccountBalance(int uid)
2     throws SQLException
3  {
4     double balance = NO_VALUE;
5     String qs = "SELECT Account.ID" +
6       "Account.Balance FROM Account;";
7     Statement stmt =
8       m_connect.createStatement();
9     ResultSet rs = stmt.executeQuery(qs);
10    while( rs.next() )
11    {
12      if( rs.getInt("id") == uid )
13      {
14         balance = rs.getDouble("balance");
15      }
16    }
17    return balance;
18 }
```

(a)

```
1  public boolean lockAccount(int card_number)
2    throws SQLException
3  {
4    boolean completed = false;
5    String qu_lock =
6      "UPDATE UserInfo SET acct_lock=1 WHERE card_number=" +
7         card_number + ";";
8    Statement update_lock = m_connect.createStatement();
9    int result_lock = update_lock.executeUpdate(qu_lock);
10   if( result_lock == 1)
11   {
12      completed = true;
13   }
14   return completed;
15 }
```

(b)

Figure 3: The `getAccountBalance` and the `lockAccount` Methods in the ATM Application.

4.4 Test Suites

Intuitively, the state of a database-driven application includes the values of all of the program's live variables and the complete state of the relational database(s) with which the program interacts. The state of a single relational database consists of all of the records within all of the relations stored inside of the database. Definition 4 states our understanding of a test suite T that can be used to assess the quality of a database-driven application [13]. We use Δ_f to denote the externally visible state of the database-driven application under test.

Definition 4. A *test suite* T is a triple $\langle \Delta_0, \langle T_1, \dots, T_e \rangle, \langle \Delta_1, \dots, \Delta_e \rangle \rangle$, consisting of an initial external test state, Δ_0, a test case sequence $\langle T_1, \dots, T_e \rangle$ for state Δ_0, and expected external test states $\langle \Delta_1, \dots, \Delta_e \rangle$ where $\Delta_f = T_f(\Delta_{f-1})$ for $f = 1, \dots, e$. □

Definition 5 notes that a specific test $T_f \in \langle T_1, \dots, T_e \rangle$ can be viewed as a sequence of test operations that cause the application under test to enter into states that are only visible to T_f. In Definition 6, we describe a restricted type of test suite where each test case returns the application under test back to the initial state, Δ_0, before it terminates. If a test suite T is not independent, we do not place any restrictions upon the $\langle \Delta_1, \dots, \Delta_e \rangle$ produced by the test cases and we simply refer to it as a non-restricted test suite.

Definition 5. A *test case* $T_f \in \langle T_1, \dots, T_e \rangle$, is a triple $\langle \delta_0, \langle o_1, \dots, o_g \rangle, \langle \delta_1, \dots, \delta_g \rangle \rangle$, consisting of an initial internal test state, δ_0, a test operation sequence $\langle o_1, \dots, o_g \rangle$ for state δ_0, and expected internal test states $\langle \delta_1, \dots, \delta_g \rangle$ where $\delta_h = o_h(\delta_{h-1})$ for $h = 1, \dots, g$. □

Definition 6. A test suite T is *independent* if and only if for all $\gamma \in \{1, \dots e\}$, $\Delta_\gamma = \Delta_0$. □

Non-restricted test suites can capture more of an application's interaction with a database while requiring the constant monitoring of database state and the potentially frequent re-computations of test adequacy. While independent test suites capture only a program's interaction with a fixed database, the measurement of the adequacy for these test suites does not require database monitoring facilities, the re-creation of an application's representation, and the re-computation of test adequacy. In Section 8, we directly address the practicality of measuring the adequacy of independent test suites and we highlight the issues associated with calculating the adequacy of non-restricted test suites.

5. DATAFLOW INFORMATION

5.1 Preliminaries

As noted by Daou et al., a database interaction point I_r can be viewed as interacting with different entities of a relational database, depending upon the granularity with which we view the interaction [6]. That is, we can view a SQL statement's interaction with a database at the level of databases, relations, records, attributes, or attribute values. Suppose that program P interacts with arbitrary database D_i at a database interaction point I_r. Database D_i can be viewed as a set of relations so that $D_i = \{r_1, \dots, r_m\}$. Moreover, an arbitrary relation r_j can be seen as a set of records such that $r_j = \{t_1, \dots, t_o\}$. Each record is an ordered set of attribute values so that $t_k = \langle t_1^k, \dots, t_q^k \rangle$. We use the notation t_l^k to denote the value of the lth attribute of the kth record in a specified relation. We use a notational convention $name(D_i)$, $name(r_j)$, $name(t_k)$ to respectively denote the name of database D_i, relation r_j, and record t_k when we want to differentiate between the names of these entities and their contents. Finally, the notational convention $name(A_l)$ and $name(t_l^k)$ are used to provide a name that respectively associates a specific attribute and attribute value with the containing relation. Definition 7 reflects our understanding of the type of a database interaction.

Definition 7. The *type* of database interaction point I_r is assumed to be *defining*, *using*, or *defining-using*. If I_r corresponds to the execution of one of the SQL operations **delete** or **update**, then I_r can be of type defining or defining-using. If I_r is the execution of the SQL **insert** statement, then I_r is of type defining. If I_r corresponds to the execution of the SQL **select** operation, then I_r is of type using. □

The defining-using type is observed at a database interaction point when the SQL **delete** and **update** statements require the deletion or insertion of records that are chosen by using other attribute values. Thus, the form of the predicate Q in the SQL **delete** and **update** statements will determine whether a database interaction point is of type defining or of type defining-using. However, at coarse views of

a database interaction, the defining-using type is subsumed by the defining type. For example, suppose that we view the SQL **delete** statement in database interaction point I_r at the coarse level of the database. Even if the **delete** defines records in one relation and the predicate Q uses an attribute in another relation, the overall operation can be seen as defining the database. Yet, when the same statement is viewed at the level of attribute interactions, it is clear that the **delete** defines the attributes in one relation and uses the attributes that are merely referenced in Q.

The call to `executeQuery` in `getAccountBalance` and the invocation of `executeUpdate` in the `lockAccount` operation are the only database interaction points in our example methods. Since the `getAccountBalance` method executes a SQL **select** statement that can only reference values within a relational database, the operation contains a database interaction point of type using. However, since the `lockAccount` method executes a SQL **update** statement that can change values within a relational database, this operation contains an interaction point that is either of type defining or type defining-using. If the interaction point in `lockAccount` is viewed at the coarse level of the database, then it is of type defining. Yet, when viewed at the level of database attributes, this interaction point uses `card_number` and defines `acct_lock`.

5.2 Enumerating Database Entities

In order to compute the database interaction associations that capture a program's interaction with a database at different levels of granularity, we provide equations to enumerate the sets of entities that might be subject to definition or usage at database interaction point I_r. These database entity sets can be used to augment the control flow graph of a method, as described in Section 7. Our formulations can enumerate $\mathcal{D}(I_r)$, $\mathcal{R}_l(I_r)$, $\mathcal{A}(I_r)$, $\mathcal{R}_c(I_r)$, and $\mathcal{A}_v(I_r)$, the sets of databases, relations, attributes, records, and attribute values at a specific interaction point, respectively.

As observed by Dauo et al., a static analysis of a database-driven application's source code will completely reveal database entities only when the interaction point is at the level of database, relation, or attribute [6]. A knowledge of the state of the database is required to correctly enumerate the sets of records and attribute values. However, the precise enumeration of $\mathcal{D}(I_r)$, $\mathcal{R}_l(I_r)$, $\mathcal{A}(I_r)$, $\mathcal{R}_c(I_r)$, and $\mathcal{A}_v(I_r)$ is not possible if the specific database interaction point I_r is either dynamic or partially dynamic. Since we will rely upon these sets of entity names to compute dataflow information that will be used for testing purposes, we favor a conservative over-estimate of these sets whenever there is any ambiguity about set membership.

In our formulations to enumerate $\mathcal{R}_l(I_r)$, $\mathcal{R}_c(I_r)$, $\mathcal{A}(I_r)$, and $\mathcal{A}_v(I_r)$ we use the functions $G_{\mathcal{R}_l}(I_r)$, $G_{\mathcal{R}_c}(I_r, j)$, and $G_{\mathcal{A}}(I_r, j)$, respectively. These functions compute the indices of the defined or used relations, records, and attributes that are referenced at interaction point I_r. The function $G_{\mathcal{R}_l}(I_r)$ returns the indices of the relation(s) whose definition or usage is detectable by a static analysis of I_r. However, if I_r is dynamic or partially dynamic, $G_{\mathcal{R}_l}(I_r)$ returns all relation indices that might be defined or used at the provided database interaction point. The function $G_{\mathcal{A}}(I_r, j)$ returns all of the attribute indices that are defined or used by I_r when it refers to relation r_j. If a static analysis of I_r does not completely reveal all of the referenced attributes, $G_{\mathcal{A}}(I_r, j)$ returns all possible attribute indices. Finally, $G_{\mathcal{R}_c}(I_r, j)$ returns the indices of all the records that are either defined or used at interaction point I_r when a reference to relation r_j occurs. The computation of $G_{\mathcal{R}_c}(I_r, j)$ requires an inspection of the state of the relation r_j. If the provided I_r is dynamic or partially dynamic, $G_{\mathcal{R}_c}(I_r, j)$ performs a conservative over-estimate of all possible records.

The name of the database within the set $\mathcal{D}(I_r)$ is established whenever program P executes the SQL **use** statement to select a database. Since our example system only interacts with the *Bank* database, we know that $\mathcal{D}(I_r) = \{Bank\}$ at all database interaction points. Equation 1 describes $R_l(I_r)$, the set of the names of relation(s) that might be subject to definition or usage if database interaction point I_r is a **select**, **delete**, or **update** statement. If I_r involves the execution of an **insert** statement that only interacts with a single relation, then we always have $\mathcal{R}_l(I_r) = \{name(r)\}$. For example, the `getAccountBalance` method described in Figure 3(a) has a database interaction point on line 9 so that $\mathcal{R}_l(I_r) = \{Account\}$. Furthermore, the `lockAccount` method described in Figure 3(b) has a database interaction point on line 9 where we have $\mathcal{R}_l(I_r) = \{UserInfo\}$.

$$\mathcal{R}_l(I_r) = \bigcup_{j \in G_{\mathcal{R}_l}(I_r)} name(r_j) \quad (1)$$

Equation 2 describes uses the enumeration of $\mathcal{R}_l(I_r)$ to determine the names of the attributes that are associated with each of these relation(s). At the database interaction point in the `getAccountBalance` method, we see that $\mathcal{A}(I_r) = \{Account.ID, Account.Balance\}$. Also, the database interaction point in the `lockAccount` method yields the attribute names $\mathcal{A}(I_r) = \{UserInfo.acct_lock, UserInfo.card_number\}$.

$$\mathcal{A}(I_r) = \bigcup_{j=1}^{|\mathcal{R}_l(I_r)|} \bigcup_{l \in G_{\mathcal{A}}(I_r, j)} name(A_l) \quad (2)$$

Equation 3 provides our formulation for $\mathcal{R}_c(I_r)$, the set of records that might be defined or used at an arbitrary database interaction point. Using the instance of the relational schema provided in Figure 1, it is clear that the database interaction point in the `getAccountBalance` method produces $\mathcal{R}_c(I_r) = \{\langle 1, \ldots, 1200, 1\rangle, \ldots, \langle 5, \ldots, 125, 4\rangle\}$.[1] Since the database interaction point in the `lockAccount` method depends upon the input to this method, we must conservatively enumerate all of the records inside the *UserInfo* relation. However, if we knew that `card_number` = 1 during the execution of `lockAccount`, we could enumerate $\mathcal{R}_c(I_r) = \{\langle 1, 32142, \ldots, 0\rangle\}$.

$$\mathcal{R}_c(I_r) = \bigcup_{j=1}^{|\mathcal{R}_l(I_r)|} \bigcup_{k \in G_{\mathcal{R}_c}(I_r, j)} name(t_k) \quad (3)$$

Equation 4 describes $\mathcal{A}_v(I_r)$, the set of attribute values that might be defined or used at a database interaction point. If we use the instance of the relational schema provided in Figure 1, the database interaction

[1] For the sake of brevity, we omit the full names of users and bank accounts in our examples of $\mathcal{R}_c(I_r)$ and $\mathcal{A}_v(I_r)$.

point in the `getAccountBalance` method yields $\mathcal{A}_v(I_r) = \{1, 1200, 1, \ldots, 5, 125, 4\}$. Once again, the input-dependent database interaction point in `lockAccount` forces the enumeration of all of the attribute values within the records of the *UserInfo* relation.

$$\mathcal{A}_v(I_r) = \bigcup_{j=1}^{|R_l(I_r)|} \bigcup_{\substack{k \in \\ G_{\mathcal{R}_c}(I_r, j)}} \bigcup_{\substack{l \in \\ G_{\mathcal{A}}(I_r, k)}} name(t_l^k) \quad (4)$$

6. TEST ADEQUACY CRITERIA

6.1 Traditional Definition-Use Associations

Throughout our discussion of a new family of dataflow-based test adequacy criteria for database-driven applications, we will adhere to the notation initially proposed in [8, 16]. For a standard program, the occurrence of a variable on the left hand side of an assignment statement is called a *definition* of this variable. The *usage* of a variable occurs when it appears on the right hand side of an assignment statement or in the predicate of a conditional logic statement or an iteration construct [11].

We will view a method in an application as a control flow graph (CFG) $G = (N, E)$ where N is the set of CFG nodes and E is the set of CFG edges. Next, we define a *definition clear path* for variable x as a path in a CFG $\langle n_\rho, \ldots, n_\tau \rangle$, such that none of the nodes $n_\rho, \ldots, n_\tau$ contain a definition or undefinition of program variable x [8]. Furthermore, we define the *def-use association* as a triple $\langle n_d, n_{use}, x \rangle$ where a definition of variable x occurs in node n_d and a use of x occurs in node n_{use} [11]. A *complete path* is a path in a method's control flow graph that starts at the CFG's entry node and ends at its exit node [8]. A complete path π_x *covers* a def-use association if it has a definition clear subpath, with respect to x and the method's CFG, that begins with node n_d and ends with node n_{use} [8].

6.2 Database Interaction Associations

While traditional definition-use associations are related to the variables in the program under test, this paper focuses on database interaction associations that involve the entities within a relational database. For an arbitrary method m, we define $\mathcal{D}(I) = \bigcup_{r=1}^{s} \mathcal{D}(I_r)$ as the set of the one or more databases that are interacted with during the execution of m. We define $\mathcal{R}_l(I)$, $\mathcal{A}(I)$, $\mathcal{R}_c(I)$, and $\mathcal{A}_v(I)$ in an analogous fashion. A database-interaction association is a triple $\langle n_d, n_{use}, x \rangle$ where the definition of relational database entity x happens in node n_d and a use occurs in node n_{use}. However, each database-interaction association is defined for a relational database entity x that is a member of one of the sets $\mathcal{D}(I)$, $\mathcal{R}_l(I)$, $\mathcal{A}(I)$, $\mathcal{R}_c(I)$, or $\mathcal{A}_v(I)$. While the database-interaction association is similar to a traditional def-use association, it does have additional semantics that are different from def-use associations for program variables.

As long as the source code of an application does not change, traditional definition-use associations remain static throughout the testing process. If a test suite T is independent, the database interaction associations that measure the suite's adequacy depend only upon the initial test suite state, Δ_0. However, if the test suite is non-restricted, the database interaction associations and the adequacy of T depend upon all subsequent test suite states, $\langle \Delta_1, \ldots, \Delta_e \rangle$. For example, if a database interaction association corresponds to the execution of the SQL **insert** statement, the set of associations that determines test suite adequacy might change if we are viewing the application's interactions at the level of records or attribute values.

When n_d corresponds to the execution of a SQL **delete** statement, the semantics of a database-interaction association differ from the traditional understanding of a definition-use association. For example, the database-interaction association $\langle n_d, n_{use}, x \rangle$ with $x \in R_c(I)$ requires the definition and usage of a record that is stored within a specific relation of a specific database. If n_d corresponds to the execution of the SQL **delete** statement, the record x is actually removed from the database and is no longer available for usage. Therefore, we allow the usage of x on node n_{use} to correspond to the usage of a *phantom record*, or a record that once existed in a previous database state but was removed during testing. The existence of phantom records (and analogously, phantom attribute values) requires the ability to monitor the state of the database and maintain information about deleted records. Yet, the usage of phantom records ensures the creation of adequacy criteria that require test suites to execute operations that delete records from a database and then check to ensure that the records are no longer available.

6.3 Test Adequacy for Database-Driven Applications

If a method's control flow graph is augmented with information about the definition and usage of the entities within a relational database, we can produce database-interaction associations for these entities instead of (or, along with) the variables in the application. A method's interaction with a database must be explicitly included within the chosen representation of this program operation. Refer to Section 7 for a detailed discussion of the database interaction control flow graph, a novel representation for database-driven applications that will enable the production of these associations and the calculation of our adequacy criteria.

Many different measurements have been proposed to assess the adequacy, or "goodness," of a test suite [23]. The standard *all-DUs* test adequacy criterion that drives def-use testing [11] is not sufficient for our purposes because it does not capture a program's interaction with a relational database. Our family of test adequacy criteria include the *all-database-DUs*, *all-relation-DUs*, *all-attribute-DUs*, *all-record-DUs*, and *all-attribute-value-DUs* adequacy measures. Definition 8 defines the *all-database-DUs* test adequacy criterion that requires a test suite T to produce a definition clear path from each definition of a database in $D(I)$ to all subsequent uses of the same database.

Definition 8. A test suite T for database interaction control flow graph $G_{DB} = (N_{DB}, E_{DB})$ satisfies the *all-database-DUs* test adequacy criterion if and only if for each association $\langle n_d, n_{use}, x \rangle$, where $x \in D(I)$ and $n_d, n_{use} \in N_{DB}$, there exists a test in T to create a complete path π_x in G_{DB} that covers the association. □

Due to space restrictions, we do not formally define the *all-relation-DUs*, *all-attribute-DUs*, *all-record-DUs*, and *all-attribute-value-DUs* test adequacy criteria. However, each of these test criterion could be defined by simply substituting $D(I)$ in Definition 8 for one of the sets $\mathcal{R}_l(I)$, $\mathcal{A}(I)$, $\mathcal{R}_c(I)$,

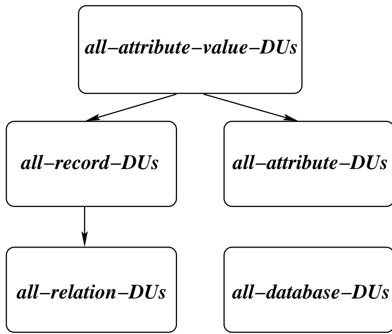


Figure 4: Test Adequacy Subsumption Hierarchy.

or $\mathcal{A}_v(I)$. In this paper, we relate our family of adequacy criteria to the control flow graph of a method. However, it is possible to associate each test adequacy criterion with a "database enhanced" version of a class control flow graph (CCFG) [9], an entire interprocedural control flow graph (ICFG) [10], or other CFG-based program representations.

6.4 Subsumption of Test Adequacy Criteria

A test adequacy criterion C_α *subsumes* a test adequacy criterion C_β if every test suite that satisfies C_α also satisfies C_β [16]. Figure 4 summarizes the subsumption relationships between our proposed test adequacy criteria. In this graph, nodes n_α and n_β represent adequacy criteria and a directed edge $n_\alpha \rightarrow n_\beta$ indicates that C_α subsumes C_β. If we assume that an arbitrary database D_i contains relations with at least one record and records with at least one attribute value, then it is evident that $|\mathcal{R}_l(I)| \leq |\mathcal{R}_c(I)| \leq |\mathcal{A}_v(I)|$. In a relational database instance with non-empty entities, we can conclude that *all-attribute-value-DUs* subsumes *all-record-DUs*, *all-record-DUs* subsumes *all-relation-DUs*, and transitively, *all-attribute-value-DUs* subsumes *all-relation-DUs.* Also, we know that *all-attribute-value-DUs* subsumes *all-attribute-DUs.* Since any database D_i can have an arbitrary number of relations, there is no subsumption relationship between *all-database-DUs* and *all-relation-DUs*, *all-record-DUs*, and *all-attribute-value-DUs*, respectively

7. A UNIFIED REPRESENTATION

7.1 Overview of Representation Construction

Intuitively, the database interaction control flow graph is a traditional CFG that contains transfers of control to one or more database interaction graphs (DIGs). It is possible for method m to contain multiple database interaction points. Moreover, each interaction point can be viewed at five different levels of granularity. Since we allow the DICFG to represent a program's interaction with a database at multiple levels of granularity, the construction of a database interaction control flow graph for method m can involve the integration of multiple DIGs for each interaction point. As our code examples in Figure 3 indicate, database interaction points are often simple assignment statements. Yet, it is also possible for a database interaction point to occur in the predicate of a conditional logic statement or an iteration construct. For simplicity, we will strictly consider database interaction points that occur in assignment statements or conditional logic predicates.[2] In order to preserve the semantics of program P, our formation of a DICFG for method m will integrate a DIG for database interaction point I_r before the node in G where the interaction actually occurs.

We can use the equations described in Section 5.2 to augment the control flow graph for method m in program P with information about the method's interaction with a relational database. In the algorithms designed to construct a DICFG, we use the sets $pred(n_\tau) = \{n_\rho | (n_\rho, n_\tau) \in E\}$ and $succ(n_\rho) = \{n_\tau | (n_\rho, n_\tau) \in E\}$ to denote the set of predecessors and successors of node n_τ and n_ρ, respectively. For an arbitrary m, a DICFG is constructed through the separate usage of the *CreateDIG* algorithm for each database interaction point in m and all the desired levels of interaction granularity. A single DICFG represents all the definitions and uses of relational database entities that could occur during the execution of m and all of the definitions and uses of the variables in method m.

7.2 Representation Details

Each database interaction point in method m corresponds to a specific node within control flow graph G. Definition 9 describes the notion of a location for any database interaction point I_r. To ensure the ability to create a DICFG that expresses a method's interaction with relational database entities at multiple levels of granularity, we use the database granularity marker $L \in \{\mathbf{D}, \mathbf{R_l}, \mathbf{A}, \mathbf{R_c}, \mathbf{A_v}\}$. We use L to denote whether an arbitrary DIG describes an interaction at the granularity of database, relation, attribute, record, or attribute value, respectively. Definition 10 defines the database interaction graph (DIG) that we use to represent an arbitrary interaction point.

Definition 9. The *location* of a database interaction point I_r in method m's CFG $G = (N, E)$ corresponds to the node $n_r \in N$. $\square$

Definition 10. A *database interaction graph* for method m's database interaction point I_r is a four-tuple $G_r = \langle N_r, E_r, I_r, L \rangle$, where N_r is a set of nodes, E_r is a set of edges, I_r is the interaction point, and L is the granularity marker. For each $n \in N_r$ we require that $|pred(n)| = 1$ and $|succ(n)| = 1$. Each G_r must have nodes $entry_r$ and $exit_r$ to distinguish the unique entry and exit points of G_r. $\square$

CreateDIG uses the *CreateDIGNode* algorithm to construct a node for each of the database entities within the input set *Name* that is enumerated using the equations provided in Section 5.2.[3] If I_r is of type defining or using, *CreateDIGNode* simply constructs a node that defines or uses entity x, respectively. However, if I_r is of type defining-using, *CreateDIGNode* must inspect I_r in order to determine whether it should construct a defining or using node for x. Each invocation of *CreateDIG* must be accompanied by a database granularity marker L in order to accommodate the calculation of test requirements for any of our adequacy

[2]It is possible to produce a DICFG for an interaction that occurs in the predicate of an iteration construct. We could integrate a DIG before the iteration construct begins and at the end of the construct's body. Since the inclusion of these constructs obscures the essence of our approach, we omit them.

[3]Due to space limitations and the peripheral nature of *CreateDIGNode*, we omit a listing of this algorithm.

Algorithm *CreateDIG*$(Name, I_r, L)$
Input: Set of Database Entity Names $Name$;
Database Interaction Point I_r;
Desired Level of Database Interaction Granularity L
Output: Database Interaction Graph G_r
1. $N_r \leftarrow N_r \cup entry_r$
2. $n_p \leftarrow entry_r$
3. **for** $x \in Name$
4. **do** $n_c \leftarrow CreateDIGNode(I_r, x)$
5. $E_r \leftarrow E_r \cup \{(n_p, n_c)\}$
6. $N_r \leftarrow N_r \cup \{n_c\}$
7. $n_p \leftarrow n_c$
8. $N_r \leftarrow N_r \cup \{exit_r\}$
9. $E_r \leftarrow E_r \cup \{(n_p, exit_r)\}$
10. **return** $\langle N_r, E_r, I_r, L\rangle$

Figure 5: The *CreateDIG* Algorithm.

Algorithm *CreateDICFG*(G, DIG)
Input: Traditional CFG G;
Set of Database Interaction Graphs DIG
Output: DICFG G_{DB}
1. **for** $G_r \in DIG$
2. **do** $I'_r \leftarrow I_r \in G_r$
3. $E'_r \leftarrow E_r \in G_r$
4. $n_r \leftarrow location(I'_r)$
5. $n_p \leftarrow pred(n_r)$
6. $E \leftarrow E \cup \{(n_p, entry_r \in E'_r\}$
7. $E \leftarrow E \cup \{(exit_r \in E'_r, n_r)\}$
8. **return** (G, DIG)

Figure 6: The *CreateDICFG* Algorithm.

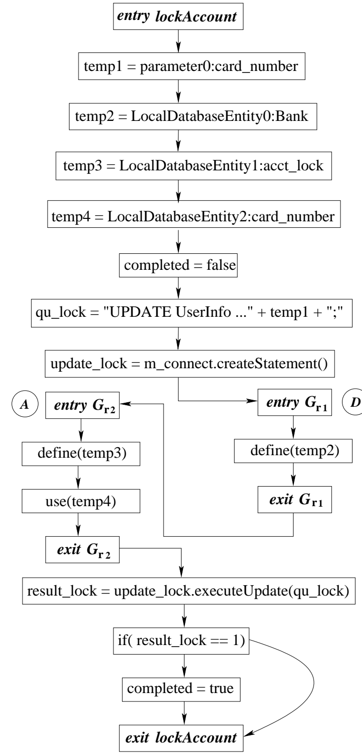


Figure 7: A DICFG for `lockAccount`.

criterion from a single DICFG. Suppose that method m contains a set of database interaction points $I = \{I_1, I_2, \ldots, I_s\}$ and the set DIG contains all of the database interaction graphs for method m. Definition 11 formally defines our understanding of a database interaction control flow graph (DICFG).

Definition 11. A *database interaction control flow graph* is a two-tuple $G_{DB} = (G, DIG)$, where G is a control flow graph for method m and $DIG = \cup\{\langle N_r, E_r, I_r, L\rangle | r \in [1, s], L \in \{\mathbf{D}, \mathbf{R_l}, \mathbf{A}, \mathbf{R_c}, \mathbf{A_v}\}\}$. □

Figure 6 provides the *CreateDICFG* algorithm for iteratively constructing our representation of an arbitrary method that interacts with a relational database. The *CreateDICFG* algorithm is responsible for extracting the location of the database interaction that each $G_r \in DIG$ represents and connecting each G_r to G at this location. The `lockAccount` operation's DICFG that includes interactions at the database and attribute level is depicted in Figure 7. Our DICFGs include nodes to define temporary variables that provide initial values for the formal parameters of methods [7]. Furthermore, we treat the entities within the relational database as global variables and insert temporaries to initially define the desired entities within the relational database [7].

8. EMPIRICAL STUDY

8.1 Experiment Goals and Design

We conducted an empirical study to investigate the practicality of computing our family of test adequacy criteria for the test suites of database-driven applications. For the purposes of this study, we manually modified the source code of two case study applications so that their source contained the desired database interaction associations. Next, we used an exhaustive intraprocedural dataflow analysis to collect the number of definition-use and database-interaction associations in the individual methods of each case study application. We also counted the number of *hanging definitions* of program variables and database entities. These hanging definitions correspond to a program variable or database entity that was defined and never involved in a subsequent use. For the purposes of this study, we assumed the existence of a hypothetical test suite T that was independent.

We empirically determined the time and space overhead incurred during the enumeration of all the intraprocedural definition-use and database-interaction associations in our case study applications. To this end, we measured the wall clock time required to compute associations at the level of no database interaction (i.e. just program variables) and each additional level of database interaction. Furthermore, we measured the average and maximum number of nodes and edges found in the CFGs of the applications that were instrumented at the levels of no database interaction and all of the additional levels of database interaction. All experiments were conducted on a GNU/Linux workstation with kernel 2.4.18-14smp, dual 1 GHz Pentium III Xeon processors, 1 GB of main memory, and a SCSI disk subsystem.

We used Soot 1.2.5 [19] to implement a tool that can calculate the intraprocedural definition-use associations, database interaction associations, and hanging definitions for all of the methods in the selected applications. For the purposes of this empirical study, our tool used the Jimple intermediate representation [19] to analyze the methods in the candidate applications.

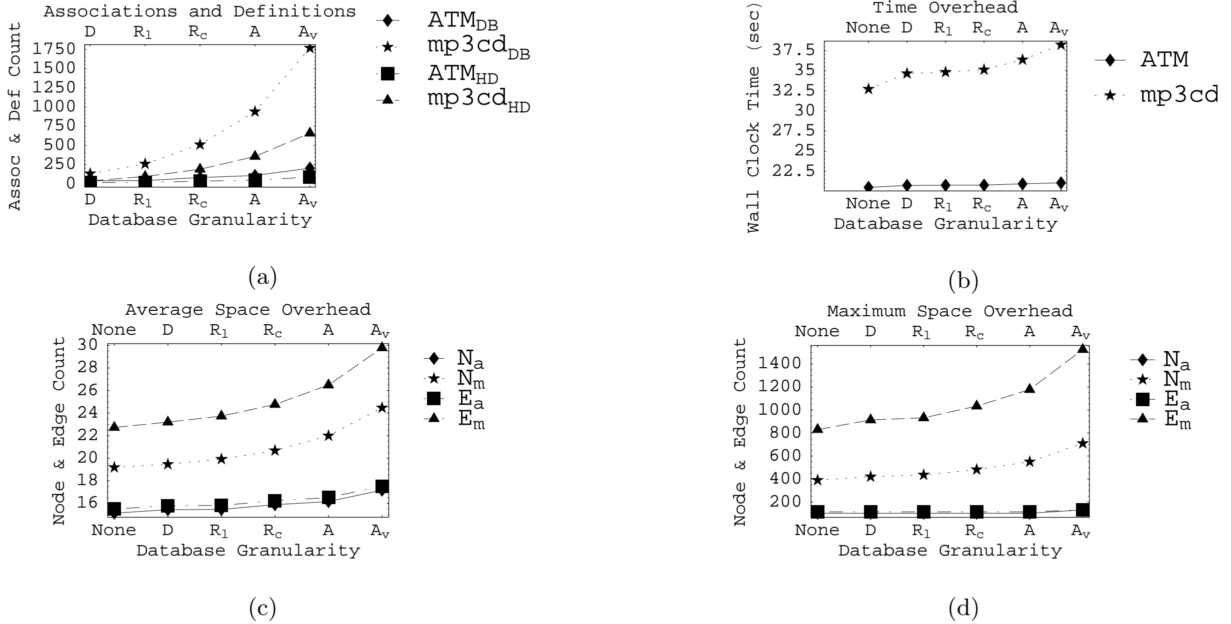


Figure 8: Empirical Study Results.

8.2 Case Study Applications

We selected two case study applications that were written in the Java programming language and use the MySQL relational database management system. The first application, `ATM`, is an implementation of the example system described in Section 2. According to JavaNCSS [1], `ATM` contains 1732 non-commented source statements, not including the MySQL driver that it uses to make connections to the database. This application contained 135 concrete methods that were subject to analysis. Furthermore, `ATM` interacts with a database that contains two relations that were each seeded with two records. The second application, `mp3cdbrowser`, is provided for download at `http://mp3cdbrowser.sourceforge.net/` and is designed to manage a local collection of mp3 files (for brevity, we refer to this application as `mp3cd`). According to JavaNCSS, `mp3cd` contains 2432 non-commented source statements, not including its MySQL driver. This application also uses the Java bytecode of a mp3 manipulation API. The mp3 manipulation bytecodes and the complete `mp3cd` application required the analysis of 452 methods. Finally, `mp3cd` interacts with a relational database that contains seven relations that were each seeded with two records.

8.3 Results Analysis

Figure 8 provides the results of our empirical study. Figure 8(a) describes the counts of database interaction associations at the level of database (D), relation (R_l), record (R_c), attribute (A), and attribute value (A_v), respectively. This graph does not include a data point for the 8718 def-use associations and 1070 hanging definitions associated with the `mp3cd` application and the 1910 def-use associations and 121 hanging definitions for `ATM`. `mp3cd`'s 1768 database interaction associations due to attribute value interactions represent 16.8% of the total number of variable and database entity associations. `ATM`'s 203 database interaction associations resulting from attribute value interactions represent 9.6% of the total number of def-use and database interaction associations. Thus, our test adequacy criteria require test suites to exercise an additional set of associations that would be neglected by traditional def-use testing. At the level of attribute value interaction, `ATM`'s 88 hanging definitions represent approximately 30.2% of the associations and definitions related to database entities. Finally, `mp3cd`'s 659 hanging definitions at the attribute level are 27.2% of the application's total number of associations and definitions. This suggests that broadening the scope of our dataflow analysis (by using a class control flow graph, for example) is likely to produce a greater number of database interaction associations and better represent the database interactions of a database-driven application.

Figure 8(b) describes the time overhead associated with the computation of intraprocedural def-use associations, database interaction associations, and hanging definitions. Each data point in this graph represents the average wall clock time for five executions of the prototype tool on each candidate application. When the `mp3cd` application contained attribute value interactions, the intraprocedural analysis took approximately 5 more seconds to complete than the same analysis on a version of `mp3cd` that contained no database interactions (x-axis label None). Furthermore, the analysis of `ATM` at the level of attribute value interaction demonstrated a very minimal overhead when compare to the analysis of `ATM` that included no database interactions. Thus, the database interaction associations that are at the heart of our adequacy criteria can be enumerated with acceptable time overhead.

Figure 8(c) and Figure 8(d) characterize the tool's space consumption by measuring the average and maximum number of nodes and edges in the original CFG and the DICFG at each interaction level. The maximum node count for `mp3cd` almost doubles and the average node count increases by approximately 8 nodes when we examine the progression from a traditional CFG to a DICFG with attribute value in-

teractions. The ATM application demonstrates a less marked increase in the average and maximum number of nodes and edges. Since an intraprocedural dataflow analysis can be efficiently conducted on our DICFGs, we can easily compute the adequacy of independent test suites. Moreover, these results place an upper bound on the time and space overhead that would be associated with each of the frequent re-computations required by non-restricted test suites.

9. RELATED WORK

Even though a significant amount of research has focused on the testing and analysis of programs, there is a relative dearth of work that specifically examines the testing of database-driven applications. While Jin and Offutt do highlight test adequacy criteria that incorporate a program's interaction with its environment [12], these authors do not specifically address the challenges associated with test adequacy criteria for database-driven applications. In [4, 5], Chays et al. describe a tool that populates databases with data that satisfies the schema constraints in a fashion that is reminiscent of the category-partition method. Unlike our family of program-based test adequacy criteria, this approach measures adequacy with respect to the specification of the database and the program and additional testing heuristics. While our family of adequacy criteria could serve as the foundation for automated test data generation, the approach described by Chays et al. will always require tester intervention.

Neufeld et al. and Zhang et al. describe similar approaches that attempt to generate database states that are consistent with respect to the constraints in the relational schema [15, 22]. Yet, neither of these approaches explicitly include a test adequacy criterion designed to isolate defects in the programs that interact with databases. While Dauo et al. propose a regression testing technique for database-driven applications in [6], their exploration of data flow issues does not include a discussion of the representation of a database-driven application or a formal understanding of a database interaction association. Chan et al. explore white-box test case generation techniques that transform SQL statements into general purpose programming language constructs in [2, 3]. However, this approach describes a program-based test adequacy criteria that focuses on a program's control flow graph and ignores potentially important dataflow information.

10. CONCLUSION

In this paper, we address the fundamental issue of test suite adequacy for database-driven applications. Our dataflow-based family of test adequacy metrics are particularly tailored to capture a program's interaction with a database at multiple levels of granularity and to ensure the existence of test suites that can detect type (1) violations of database validity and completeness. Our empirical studies indicate that it is possible to compute the intraprocedural database interaction associations in a DICFG with a minimal time and space overhead. These studies also confirm that a significant number of important database interactions will be overlooked by traditional def-use testing. In future research, our family of test adequacy criteria can serve as the foundation for tools that focus on automated test generation and regression testing for database-driven applications.

11. REFERENCES

[1] JavaNCSS. `http://www.kclee.com/clemens/java/javancss/`.

[2] Man-Yee Chan and Shing-Chi Cheung. Applying white box testing to database applications. Technical Report HKUST-CS9901, Hong Kong University of Science and Technology, Department of Computer Science, February 1999.

[3] Man-Yee Chan and Shing-Chi Cheung. Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374, March 1999.

[4] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weyuker. A framework for testing database applications. In *Proceedings of the 7th International Symposium on Software Testing and Analysis*, pages 147–157, August 2000.

[5] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. AGENDA: A test generator for relational database applications. Technical Report TR-CIS-2002-04, Department of Computer and Information Sciences, Polytechnic University, Brooklyn, NY, August 2002.

[6] Bassel Daou, Ramzi A. Haraty, and Nash'at Mansour. Regression testing of database applications. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 285–289. ACM Press, 2001.

[7] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the 18th International Conference on Software Engineering*, pages 575–584. IEEE Computer Society Press, 1996.

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

[9] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163. ACM Press, 1994.

[10] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, 1994.

[11] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[12] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification & Reliability*, 8(3):133–154, 1998.

[13] Atif M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, University of Pittsburgh, Department of Computer Science, 2001.

[14] Amihai Motro. Integrity = validity + completeness. *ACM Transactions on Database Systems*, 14(4):480–502, 1989.

[15] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. Generating consistent test data: Restricting the search space by a generator formula. *VLDB Journal*, 2:173–213, 1993.

[16] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4), April 1985.

[17] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Companies, Inc., New York, NY, 2002.

[18] Diane M. Strong, Yang W. Lee, and Richard Y. Wang. Data quality in context. *Communications of the ACM*, 40(5):103–110, 1997.

[19] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[20] Yair Wand and Richard Y. Wang. Anchoring data quality dimensions in ontological foundations. *Communications of the ACM*, 39(11):86–95, 1996.

[21] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–76, January/February 2000.

[22] Jian Zhang, Chen Xu, and S.C. Cheung. Automatic generation of database instances for whitebox testing. In *Proceedings of the 25th Annual International Computer Software and Applications Conference*, October 2001.

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