Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Testing Database Transaction Concurrency

Yuetang Deng   Phyllis Frankl   Zhongqiang Chen

Department of Computer and Information Science

Technical Report
TR-CIS-2003-03
10/20/2003

Testing Database Transaction Concurrency[1]

Yuetang Deng Phyllis Frankl Zhongqiang Chen
Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201 USA
{ytdeng, phyllis, zchen}@cis.poly.edu

Abstract

Database application programs are often designed to be executed concurrently by many users. By grouping related database queries into transactions, DBMS systems can guarantee that each transaction satisfies the well-known ACID properties: Atomicity, Consistency, Isolation, and Durability. However, if a database application is decomposed into transactions in an incorrect manner, the application may fail when executed concurrently due to potential offline concurrency problems.

This paper presents a dataflow analysis technique for identifying schedules of transaction execution aimed at revealing concurrency faults of this nature, along with techniques for controlling the DBMS or the application so that execution of transaction sequences follows generated schedules. The techniques have been integrated into AGENDA, a tool set for testing relational database application programs. Preliminary empirical evaluation is presented.

1. Introduction

Database and transaction processing systems occupy a central position in our information-based society. It is essential that these systems function correctly and provide acceptable performance. Substantial effort has been devoted to insuring that the algorithms and data structures used by Database Management Systems (DBMSs) work efficiently and protect the integrity of the data. However, relatively little attention has been given to developing systematic techniques for assuring the correctness of the related database application programs. Given the critical role these systems play in modern society, there is clearly a need for new approaches to assess the quality of the database application programs.

To address this need, we have developed a systematic, partially-automatable approach and a tool set based on this approach to testing database applications [4,5,6] Initially, we focused on problems associated with individual queries. In [8] we explore problems dealing with transactions, comprising groups of related queries. This paper focuses on concurrency problems associated with multiple simultaneously running instances of an application program.

Many database applications are designed to support concurrent users. For example, in an airline reservation system, many clients can simultaneously purchase tickets. To avoid simultaneous updates of the same data item and other potential concurrency problems, database applications employ program units called transactions, consisting of a group of queries that access, and possibly update, data items. The DBMS schedules query executions so as to insure data integrity at transaction boundaries while trying to utilize resources efficiently.

Although the DBMS employs sophisticated mechanisms to assure that transactions satisfy the ACID properties – Atomicity, Consistency, Isolation, and Durability – an application can still have concurrency-related faults if the application programmer erroneously placed queries in separate transactions when they should have been executed as a unit. The focus of this paper is on detecting faults of this nature.

The rest of the paper is organized as follows. Section 2 introduces the background, including the AGENDA tool set for database application testing. Section 3 identifies some typical transaction concurrency problems. Our approach to test for concurrency problems is presented in Section 4. Section 5 discusses the problems with transaction atomicity/durability in database applications and our solution. Section 6 presents preliminary empirical evaluation. Section 7 describes the related work. Section 8 concludes with our contributions and discusses future work.

2. Background

2.1. Database Transaction

A transaction is a means by which an application programmer can package together a sequence of database operations so that the database system can provide a number of guarantees, known as the ACID properties of a

[1] Supported in part by NSF grant CCR-9988354

transaction [11]. Atomicity ensures that the set of updates contained in a transaction must succeed or fail as a unit. Consistency means that a transaction will transform the database from one consistent state to another. The isolation property guarantees that partial effects of incomplete transactions should not be visible to other transactions. Durability means that effects of a committed transaction are permanent and must not be lost because of later failure. Therefore, transactions can be used to solve problems such as creation of inconsistent results, concurrent execution errors within transactions and lost results due to system crash [16].

A database application program typically consists of SQL statements embedded in a source language, such as C or Java. Ends of transactions are delimited by the keyword COMMIT (or ROLLBACK). Once a transaction begins execution, the DBMS assures that no other transaction executes a statement that interferes with this transaction (e.g. by updating data being accessed by the transaction) until the transaction commits or rolls back. The system can initiate a rollback when it enters a state in which ACID properties may be violated, either due to overly optimistic scheduling decisions or to external events such as system failure. When a rollback occurs, the system is restored to the state before execution of the transaction. Once a transaction commits, all of the changes it has made to the database state become permanent.

In this paper, we consider application programs written in a high-level language such as C, with SQL statements interspersed. The SQL statements may involve *host variables*, which are variables declared in the host program (host variables are preceded by colons). Multiple clients may execute an application program concurrently. In this case, each client has its own host variables. The only "variable" shared by concurrently executing clients is the database. In this paper we do not consider stored procedures or shared variables arising through multi-threading within an application instance.

2.2. AGENDA Tool Set

In our previous work [4,5], we have discussed issues arising in testing database systems, presented an approach for testing database applications, and described AGENDA, a set of tools based on this approach. In testing such applications, the states of the database before and after the application's execution play an important role, along with the user's input and the system output. A framework for testing database applications was introduced. A complete tool set based on the framework has been prototyped. The components of this system are: AGENDA Parser, State Generator, Input Generator, State Validator, and Output Validator.

AGENDA takes as input the database schema of the database on which the application runs, the application source code, and "sample value files", containing some suggested values for attributes. The tester interactively selects test heuristics and provides information about expected behaviors of test cases. Using this information, AGENDA populates the database, generates inputs to the application, executes the application on those inputs and checks some aspects of correctness of the resulting database state and the application output.

This approach is loosely based on the Category-Partition method [17]: the user supplies suggested values for attributes, optionally partitioned into groups, which we call data groups. The data are provided in the sample value files. The tool then produces meaningful combinations of these values in order to fill database tables and provide input parameters to the application program. Data groups are used to distinguish values that are expected to result in different application behaviors. For example, in a payroll system, different categories of employees may be treated differently. Additional information about data groups, such as probability for selecting a group, can also be provided via sample value files.

Using these data groups and guided by heuristics, AGENDA produces a collection of test templates representing abstract test cases. The tester then provides information about the expected behavior of the application on tests represented by each test template. For example, the tester might specify that the application should increase the salaries of employees in the "faculty" group by 10% and should not change any other attributes. In order to control the explosion in the number of test templates and to force the generation of particular kinds of templates, the tester selects heuristics. Finally, AGENDA instantiates the templates with specific test values, executes the test cases and checks that the output and new database state are consistent with the expected behavior indicated by the tester. The architecture of AGENDA is shown in Figure 1.

The first component (AGENDA Parser) extracts relevant information from the application's database schema, the application queries, and tester-supplied sample value files, and then makes this information available to the other four components by creating an internal database called AGENDA DB to store the extracted information. The AGENDA DB is used and/or modified by the remaining four components. AGENDA Parser is a modified version of from the open-source DBMS Postgresql parser [18].

The second component (State Generator) uses the database schema along with information from the tester's sample value files indicating useful values (optionally partitioned into different groups of data) for attributes, and populates the database tables with data satisfying the

integrity constraints. It retrieves the information about the application's tables, attributes, constraints, and sample data from the AGENDA DB and generates an initial DB state for the application. We refer to the generated DB as the application DB. Heuristics are used to guide the generation of both the application DB state and application inputs.

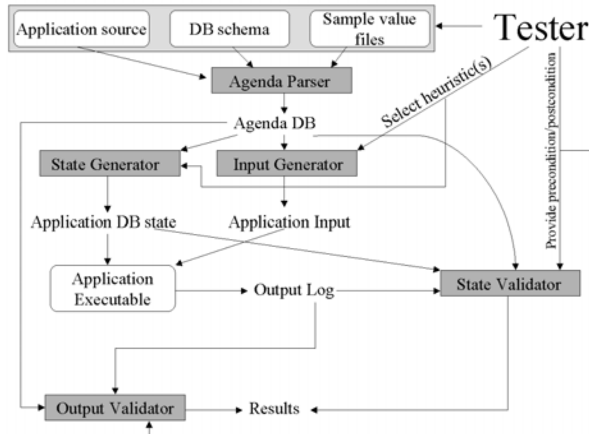

Figure 1: Architecture of the AGENDA tool set

The third component (Input Generator) generates input data to be supplied to the application. The input data are created by using information generated by the AGENDA Parser and State Generator components, along with information derived from parsing the SQL statements in the application program and information that is useful for checking the test results.

The fourth component (State Validator) investigates how the state of the application DB changes during execution of a test. It automatically constructs a log table and triggers/rules for each table in the application schema. The State Validator uses these triggers/rules to capture the changes in the application DB automatically, and uses database integrity constraint techniques to check whether the application DB state has changed in the right way.

The fifth component (Output Validator) stores the tuples satisfying the application query and constraints in the log tables. It uses integrity constraint techniques to check that the preconditions and post-conditions hold for the output returned by the application.

3. Transaction concurrency problems

Concurrency is one of the trickiest aspects of software development. Even if an application runs correctly for all inputs in an isolated environment, it may have incorrect behavior when running concurrently with other instances of the same application. We define *concurrency failure* to be a failure that occurs when two or more instances execute concurrently, which could not occur if the instances executed serially. Transaction mechanisms in a DBMS provide protection for the shared resource (i.e., the database). However, designers must use them carefully to assure correct behavior.

The transaction manager of a DBMS schedules the execution of queries from concurrent instances. It assures that no concurrency failures occur, *provided that the application programmer uses transactions appropriately*. Without intervention by the transaction manager, the following phenomena are possible for two concurrent database transactions T^i and T^j (executed concurrently by two instances)[2, 12].

P0 (Lost update/Dirty Write): The update to a data element committed by T^i is ignored by transaction T^j, which writes the same data element based on the original value.

P1 (Dirty Read): T^i modifies a data element. T^j reads that data element before T^i commits. If T^i rolls back, T^j will use an uncommitted value which does not exist in the database.

P2 (Non-repeatable Read): T^i reads a database element twice, once before transaction T^j updates it and once after transaction T^j has updated it and commits. These two read operations return different values for the same data element.

P3 (Phantom): T^i reads a set of tuples based on some specified conditions. T^j then executes and may insert one or more tuples that satisfy the same conditions as T^i. When T^i repeats the reading, it will obtain a different set of tuples. The difference between P2 and P3 is that new tuples appear in P3 while the same tuple with different values may be read in P2.

DBMS systems allow transactions to run at four isolation levels from least stringent (highest concurrency) to most stringent (lowest concurrency) [15]: READ UNCOMMITTED (level 0), READ COMMITTED (level 1), REPEATABLE READ (level 2), SERIALIZABLE (level 3). P0 is impossible at any isolation level while P1, P2, and P3 may happen at levels 0, 1 and 2. When transactions are executed at the SERIALIZABLE level by default, the DBMS transaction manager assures that none of the four phenomena can occur. It schedules execution of the queries in the concurrent transactions in such a way that the result of the concurrent executions is the same as that of a sequential execution of transactions in some certain order.

However, phenomena analogous to P0, P1, P2, and P3 may occur in an application even if all transactions run at the SERIALIZABLE level. In this case, these phenomena occur between transactions instead of within a single transaction and they are classified as *offline concurrency* problems [10], which occur when data are manipulated

across multiple database transactions. A buggy transaction design/implementation may group queries into transactions in an incorrect manner. In the worst case, it may put each query into a separate transaction. These incorrect designs and/or implementations of database applications may cause concurrency failures. Concurrency control mechanisms in DBMS systems cannot avoid these problems.

A common scenario in a database application is that the application selects some data from a database, processes the data, and updates the database with new data. Consistency constraints can be violated if more than one instance tries to modify the same data element concurrently. To avoid interference from other instances, related operations should be grouped into a transaction. The entire application can be implemented as a transaction to avoid concurrency problems, but this would prevent any concurrency since all instances would essentially be forced to execute sequentially. To achieve some degree of concurrency, an application is composed of a set of transactions. There is a close relation between the concurrency and the number of transactions in the application. The incorrect manipulation of this relation to over-optimize efficiency easily introduces concurrency faults into the system. The examples below illustrate offline concurrency problems.

A partial and buggy implementation of an airline reservation system is shown in Figure 2. The attribute *avail* in table *Flights* represents the number of seats still available on a given flight. In line 5, the number of seats available on a given flight is stored in the host variable *avail* and is decremented by 1 in line 6 indicating that one seat was booked for the flight. The application erroneously commits between lines 5 and 6, thus allowing another concurrent application instance to modify the database between executions of these lines.

Example 1: Ticket booking system

```
CREATE TABLE Flights(fltNum INT, avail INT);
1)  BEGIN DECLARE SECTION;
2)      int     flight;
3)      int     avail;
4)  END DECLARE SECTION;
5)  void chooseSeat( ) {
        EXEC SQL SELECT avail INTO :avail
          FROM Flights WHERE fltNum = :flight
5')    COMMIT;          //Transaction T1
       // Erroneously ends T1 and begins T2
6)     if (avail > 0) {
          EXEC SQL UPDATE Flights SET
            avail =:avail -1 WHERE fltNum = :flight ;
7)     COMMIT;          // Transaction T2
    }
```

Figure 2: A sample database application system illustrating a potential offline concurrency error.

If two clients A and B run *chooseSeat()* simultaneously, the following schedule could occur: client A runs T1, client B runs T1, client A runs T2, client B runs T2. After execution of this schedule, *avail* is decreased by 1 instead of 2. It implies phenomenon P0.

Figure 3 shows a partial implementation of a warehouse application. The sum of attribute *ol_amount* in the table *customer* represents the total balance for one order. In line 1, this value is stored in the host variable *ol_total.* In line 2, *c_balance* is increased by *ol_total* indicating that this order is delivered. The application erroneously commits between line 1 and line 2, thus allowing another concurrent application instance to modify the database between the executions of these lines.

Example 2: A warehouse application

```
CREATE TABLE order_line
    (o_id INT, ol_amount MONEY );
CREATE TABLE customer
    (c_id INT, c_balance MONEY);
Delivery ( ) {
1)   SELECT SUM(ol_amount) INTO :ol_total
        FROM order_line WHERE c_id = :c_id;
2')   COMMIT;  // Transaction T3
      // Erroneously ends T3 and begins T4
2)    UPDATE customer SET c_balance =
         c_balance + :ol_total where c_id = :c_id;
3)    COMMIT;   // T4
}
New_order ( ) { ….
     INSERT INTO order_line VALUES
        (:c_id, :amount); // T5
}
```

Figure 3: Partial implementation of TPC-C warehouse application.

If two clients, A and B, run *Delivery( )* with the same *c_id* simultaneously, then the following schedule could cause the database to be inconsistent between the *order_line* and *customer* tables: client A runs T^3, client B runs T^5, client A runs T^4. This schedule implies phenomenon P0. We describe technique for analyzing and testing application programs for potential offline concurrency faults in the next section. In this paper, we only consider concurrent transactions running at the SERIALIZABLE, the most stringent level. All concurrency problems at a higher isolation level could also appear at a lower isolation levels. Additional faults could occur if the application programmer erroneously assigns a less stringent level than needed.

4. Testing for concurrency failures

Testing concurrent systems is notoriously difficult due to non-determinism and synchronization problems. Multiple executions of the same test may have different interleavings and generate different results, making it particularly difficult to diagnose and debug faults that have been detected. In addition to the problems that can occur in sequential programs, concurrent programs have their unique problems: data access control and synchronization. In a database application, there is usually no direct synchronization between different application instances. Concurrency problems may occur when two application instances simultaneously access and/or update the same database element (the same attribute of the same table) directly or indirectly through host variables storing local copies of the same database element. Based on the above observations, we propose the following procedure to test transaction concurrency.

1. Find schedules that could potentially produce concurrency failures. A schedule can be represented by the form $S = <T^i_A, T^j_B, T^k_A>$, where T^i, T^j and T^k are transactions and are not necessarily distinct[2], and A and B are two different application instances. Section 4.1 describes relevant schedules and Section 4.2 gives an algorithm for finding them.
2. Generate test cases to execute the given transactions for each generated schedule. Transactions in a schedule usually access the same database element (table and attribute). To guarantee that all transactions in a schedule access the same tuple of the same database table, the test case generator should populate the inputs to these transactions properly.
3. Execute the test cases in such a way that the interleaving of executions of application instances conforms to the specified schedule. This can be achieved by using either of the two methods: modification of the concurrency control engine in the DBMS backend and instrumentation of the application source code.

4.1. Failure-prone transaction schedules

The simplest schedule which may induce concurrency failure is one that involves only three transactions and has the form of $<T^i_A, T^j_B, T^k_A>$. A naïve approach would generate all possible triples of the form $<T^i_A, T^j_B, T^k_A>$. However, many of them are not relevant since the transactions access totally different data elements. It is more efficient if we only generate transaction sequences accessing the same data element directly or through host variable. We call such schedules *interesting*. A sequence $<T^i_A, T^j_B, T^k_A>$ is *legal* if an application instance A can follow a path containing T^i followed by T^k.

To define legal schedules, we introduce a transaction control flow graph (TCFG) based on the application source code. Each node represents a host language statement or an embedded SQL statement. There is a directed edge from node *i* to node *k* if the execution of *i* can be followed immediately by the execution of *k*. All the consecutive non-transaction nodes (those nodes consisting of host language statements only) can be collapsed together to reduce the size of the TCFG as long as the control structure of the application is not changed. The TCFG is constructed by using source code analysis tools [8]. The flow information inside a transaction is only helpful in testing transaction consistency. In this paper, the TCFG is further simplified by collapsing all SQL statements of one transaction into one single transaction node.

Each transaction node in the TCFG is associated with the set of database elements and the set of host variables the transaction accesses. This information can be obtained from the AGENDA tool set introduced in [5]. When SQL queries in a transaction are parsed by the AGENDA Parser, information about all database elements to be accessed (reads or writes) is stored in the table *xact_read_write*, and information about host variables to be accessed (defines or uses) and their associated tables and attributes is stored in the table *xact_parameter*. For instance, table *xact_read_write* for Example 1 and table *xact_parameter* for Example 2 are given in Tables 2 and Table 3.

Table 2: table xact_read_write for example 1, R stands for "READ"; W stands for "WRITE".

XactId	Op	Tname	Aname
T^1	R	Flight	Available
T^1	R	Flight	Fltnum
T^2	R	Flight	Fltnum
T^2	W	Flight	available

Table 3: table xact_parameter for example 2, U stands for "USE"; D stands for "DEFINE".

XactId	Param	Op	Tname	Aname
T^3	ol_total	D	order_line	ol_amount
T^3	c_id	U	order_line	c_id
T^4	ol_total	U	customer	c_balance
T^4	c_id	U	Customer	c_id
T^5	c_id	U	customer	c_id
T^5	amount	U	customer	c_balance

As illustrated in Example 1, if two transactions T^i, and T^k in the same instance access the same data element in some path of the TCFG, there is a potential concurrency failure: a transaction T^j in another instance could access the same data element; if their operations are not

[2] T^i , T^j , and T^k are not necessarily distinct.

compatible (i.e., at least one operation is WRITE) a concurrency failure could occur.

Similarly, as illustrated in Example 2, if two transactions T^i and T^k define/use the same host variable in some path of TCFG, and a transaction T^j in another application instance writes the data element associated with this host variable, then T^j may change the data element, so the host variable is not consistent with the associated data element. Again, a concurrency error may occur.

In Example 2 above, in T^k client A is using a host variable (defined in T^i) whose value is out of date because client B updated the data element from which the host variable is defined before client A used the host variable. More generally, T^k could use a host variable that depends on the host variable defined in T^i, through some chain of definitions and uses.

By considering all possible interleavings between a transaction T^j in one instance and transactions (T^i, T^k) from another instance, we list all the possible access patterns and possible errors with each pattern in Table 4 and Table 5.

Table 4: access patterns and their problems. Transactions (T^i, T^k) in one instance and T^j in another access the same database element.

T^i	R	W	R	**R**	**R**	**W**	**W**	**W**
T^j	R	R	R	**W**	**W**	**W**	**W**	**R**
T^k	R	R	W	**R**	**W**	**R**	**W**	**W**
Conflict	N	N	N	**Y**	**Y**	**Y**	**Y**	**Y**
Phenomena				**P2**	**P0**	**P0 P3**	**P0**	**P1**

Table 5: Access patterns and their problems. Transaction (T^i, T^k) belongs to an instance, a host variable is defined/used in T^i and T^k. T^j in another instance accesses the database element associated with the host variable in T^i.

T^i	U	D	U	**U**	**U**	**D**	**D**	D
T^j	R	R	R	**W**	**W**	**W**	**W**	R
T^k	U	U	D	**U**	**D**	**U**	**D**	D
Conflict	N	N	N	**Y**	**Y**	**Y**	**Y**	N
Phenomena				**P0 P1 P2 P3**	**P0 P1 P2 P3**	**P0 P1 P2 P3**	**P2 P3**	

4.2. Generation of interesting and legal schedules

A flow-sensitive data flow analysis technique is used to construct interesting and legal schedules. Data flow analyses (DFAs) are widely used in static program analysis [1]. In DFA, a graph is constructed for the control and/or data flow in the program. Each node represents a statement or a block of statements. Control and data information are associated with each node and can be propagated along edges. Information associated with flow graph nodes is usually iteratively re-computed by the DFA algorithm.

Each node in the TCFG that correspond to a transaction is associated with a set of tuples in form of < T^k, *op* , *elm* >, where data element *elm* is either a database element (table.attribute) or a host variable, and *op* is the operation (R/W or D/U) of transaction T^m on *elm*.

A DFA algorithm similar to the well-known reaching definitions algorithm is used to find pairs of transactions that access related data and lie on the same control path. For each node GEN and KILL sets are defined. They represent both data element flow information and host variable flow information:

$$GEN[T^k] = \{< T^k , op , elm > \mid < T^k , op , elm > \in (xact_read_write \cup xact_parameter)\}$$

$$KILL[T^k] = \{<T^j , op1 , elm > \mid <T^j , op1 , elm > \in (xact_read_write \cup xact_parameter) \wedge (k\neq j) \wedge (<T^k, op2, elm > \in GEN [T^k]) \wedge op2 \in \{W,D\} \}$$

For each node (transaction) T^k, GEN[T^k] contain a tuple for each data element *elm* accessed by T^k. For each data element *elm* that is updated in node T^k , KILL[T^k] includes all tuples involving a data element, *elm*, that is defined or written in T^k. IN[T^k] and OUT[T^k] can be calculated iteratively by using a work-list algorithm derived from the method for computing the Reaching Definitions [1], which finds all the tuples reaching node T^k by solving the following equations:

$$IN[T^k] = \cup_{Ti \in predecessors(Tk)} (OUT [T^i]).$$

$$OUT[T^k] =(IN [T^k] - KILL [T^k]) \cup GEN [T^k]$$

After the algorithm converges, IN[T^k] and GEN[T^k] can be used to derive pairs of transactions accessing the same data element in the same execution path. If a tuple <T^i*, op1, elm* > is in the IN set of T^k and <T^k*, op2, elm* > is in GEN set, then a tuple < T^i*, op1,* T^k*, op2, elm* > is added to the XactPair set indicating that transactions T^i and T^k access the same data element elm. The interesting and legal schedule XactSchedule set can be generated based on the XactPair set, table 4 and table 5. The procedures for computing XactPair set and XactSchedule set are given in Figure 5.

```
Gen_XactPair( T^k ) {
    For each tuple <T^i, op1, elm > in IN[T^k]
        If exists  tuple <T^k, op2, elm > in GEN[T^k]
            Add <T^i, op1, T^k, op2, elm > to XactPair
}
Gen_XactSchedule( ) {
    For each tuple <T^i, op1, T^k, op2, elm> in XactPair
```

If exists tuple <T^j, op3, elm > $\in$ GEN(T^j) $\wedge$
(<op1, op3, op2> has conflict)
Add <T^i_A, T^j_B, T^k_A> to the XactSchedule
}

Figure 5: Generation of transaction pairs and schedules

For instance, by applying the above algorithm to Example 1, <T^1, R, T^2, R, *flights.fltNum*> and <T^1, R, T^2, W, *flights.avail*> are added to XactPair because both T^1 and T^2 access the same data items *flights.fltNum* and *flights.avail*. Both transactions read data element *flights.fltNum* only, and no transaction write *flights.fltNum*, so no schedule is generated. For data element *flights.avail*, two schedules are generated: <$T_A{}^1$, $T^1{}_B$, $T_A{}^2$> and <$T^1{}_A$, $T^2{}_B$, $T^2{}_A$>.

Based on the same reasoning, < T^3, U, T^4, U, *c_id*> and < T^3, U, T^4, U, *ol_total* > are added to XactPair for Example 2 and schedule <$T_A{}^3$, $T_B{}^5$, $T_A{}^4$> is generated. Notice that there are no pairs XactPair < T^3, U, T^5, U, *c_id*> and < T^4, U, T^5, U, *c_id*> because T^3 (or T^4) and T^5 do not appear in the same execution path, assuming that there is no control path in the calling program that calls Delivery() then calls New_Order()[3]. It is clear that, by using flow-sensitive analysis techniques, the number of generated schedules can be reduced dramatically.

4.3. Generation and execution of test cases for schedules

The above (static) analysis is conservative, in the sense that every schedule corresponding to the phenomena in Table 4 and Table 5 is generated. Consequently, if no schedules are generated, the tester can be assured that none of the concurrency failures under consideration can occur. A schedule that is generated may or may not lead to a concurrency failure. In the remaining (dynamic) phases, we attempt to execute the anomalous schedules in order to exhibit failures. Some schedules may be infeasible because the required paths cannot be executed (due to unsatisfiable relations among variables), and some schedules may not cause concurrency failures if data values are not correlated. Also, a schedule might indicate a phenomenon (P1, P2 or P3) which is acceptable for the application.

To enforce the generated schedule (or partial schedule), test cases must be generated in such a way that transactions in the given schedule manipulate the same tuple (row) in the database table. By scrutinizing the documentation of the application design and implementation, testers can get some ideas about each transaction, and figure out which transactions will be executed for each test template. Therefore, to generate test cases, we first need to identify appropriate test templates and then instantiate them.

In Example 1, the test template for schedule <$T^1{}_A$, $T^2{}_B$, $T^2{}_A$> involves the executions of T^1 and T^2. For test case of this test template, input to $T^1{}_A$, $T^2{}_A$, and $T^2{}_B$ must be generated appropriately such that the executions of these transactions indeed follow the given schedule. To make sure that transactions operate on the same tuple in the database, some parameters (host variables) in the preconditions of these transactions must be instantiated with the same values.

Generally speaking, for a test case for a database application, the DBMS scheduler will execute the test according to its own scheduling policy, and the schedule chosen by the DBMS may not be the one in which we are interested. To guarantee that the test is executed according to the desired schedule, we suggest two methods, modification of the DBMS transaction manager and instrumentation of the application. Both methods use shared memory and semaphores for synchronization and coordination of different application instances. Shared memory is used since it is the fastest form of IPC (Inter-process Communication) and shared data does not need to be copied between processes [19].

In a DBMS system, the transaction manager monitors transaction executions, guarantees transaction ACID properties, and recovers from failure [3]. To modify the transaction manager so that it will execute our tests according to our policy, we consider two scenarios. First consider complete schedules in which the execution order of all transactions is specified. We modify the transaction manager so that it reads information about the desired schedule (e.g., from a configuration file) into its shared memory. Semaphore is used to control the access by all DBMS back-ends. Based on the connection identifier (i.e. process identifier) and transaction identifier, the transaction manager can determine if the current application instance is the desired one. If not, the manager just schedules it in a normal way. Otherwise, the manager will not schedule the instance until it is the instance's turn.

The second scenario is that only a partial schedule is generated and the execution order of a subset of transactions is specified. Different transactions may contain the same query. To distinguish transactions, we assign a static unique identifier (*xid*) for each transaction by instrumenting the following statement to the head of each transaction:

```
Select * from xact_table where xact_table.id = xid
```

By *xid*, the transaction manager can identify whether or not the current transaction should be regulated and subjected to the specified schedule. If the specified schedule is non-executable for the given test case, the

[3] The TCFG and data flow analysis are currently intraprocedural. Adapting them to handle interprocedural flows would yield more accurate analysis.

transaction manager will raise an exception and abort the test after waiting for some given time.

The second way to execute a schedule is to add control in the application source code. Similar to the first method, information about the desired schedule is read into shared memory from a configuration file. The shared memory can be accessed by all instances; however, the access is mutually exclusive via semaphores. The instrumentation tool finds the starting points of all transactions involved in the schedule, and adds a function call to wait_for_my_turn(). This function will access the scheduling information in the shared memory, and check if it is the current transaction's turn to execute. If not, the transaction yields its execution and keeps on waiting. Otherwise, the instance will execute until the end of the current transaction. The above procedure is repeated until all instances are finished. Similar techniques discussed in the previous method can be used to handle non-executable schedules.

In the first method, application instances are independent of each other and they can run on different machines. The second method is independent of the DBMS systems and has better portability and flexibility. In section 6, we will compare their overhead and efficiency.

5. Testing transaction atomicity/durability

The property of atomicity/durability is ensured by DBMS systems. The atomicity property requires that we execute a transaction to completion. If a transaction fails to complete for some reasons (e.g., system crash), the DBMS system must undo any effect the transaction imposed on the database. However, if a transaction is chopped into two or more transactions due to buggy design/implementation, then the DBMS system can not recover if the application fails after the first transaction commits. Again, this is the offline concurrency problem, which cannot be solved by the DBMS itself. Example 3 is uch a buggy implementation of balance transfer in a bank application.

```
Example 3: balance transfer application
CREATE TABLE checking
      (acctNum INT, balance INT);
CREATE TABLE saving
      (acctNum INT, balance INT);
Transfer ( ) {
  SELECT balance into :out_balance FROM
        saving WHERE acctNum = :in_acct1;
      If (:out_balance > :amount) {
        UPDATE saving SET balance = balance –
           :amount WHERE acctNum = :in_acct1;
        COMMIT;  // Transaction T6
           //  Erroneously ends T6 and begins  T7
        UPDATE checking SET balance = balance
           + :amount WHERE acctNum = :in_acct2;
        COMMIT; //T7 }
}
```

Figure 6 Partial C program for bank transfer.

To test for atomicity/durability problems, we instrument the database application source code in the following way. For each transaction in the test, we find its end point (commit or rollback), and insert code which sends a signal to AGENDA's Validator and then suspends on the pause( ) function. After verifying that database is in a consistent state, the State Validator will send a signal back to the application to resume its execution. To verify database states, the State Validator needs to know the preconditions and post-conditions of transactions. For instance, in example 3, the precondition and post-condition are that the sum of two accounts involved does not change. This knowledge is then converted into a check constraint in the log tables.

6. Preliminary evaluation

We have implemented the proposed methods and measured some aspects of their performance on the TPC-C benchmark. We performed all experiments on the platform of Sun ULTRA 10 workstation. The CPU clock rate is 440 Mhz. Main memory is 384 MB. The TPC-C application is implemented in C programming language with embedded SQL. The instrumentation is implemented in Perl. TPC Benchmark™ C (TPC-C) is the standard benchmark for online transaction processing (OLTP). It is a mixture of read-only and update-intensive transactions that simulate the activities found in a complex OLTP [20]. The TPC-C application models a wholesale supplier managing orders and stocks. The five OLTP transactions are new-orders, payment, order-status, delivery, and stock-level.

To evaluate the time overhead of the proposed methods, we chop each original transaction into 2 transactions arbitrarily. This introduces potential concurrency faults. We run the TPC-C application in three different ways and record their corresponding CPU elapse time.

S0: The original five transactions are executed without any instrumentation or scheduling enforcement.

S1: The desired schedule is enforced by the DBMS transaction manager.

S2: The transactions are instrumented in the application source code according to the desired schedule.

We ran S0, S1 and S2 separately 5 times and recorded their total elapse time in the first three rows of Table 6. The unit of time is seconds. The last two rows of Table 6

show the overhead for situations S1 and S2, defined as OHi = (Si-S0)/S0 * 100% (i=1 or 2).

As we see from Table 6, the overhead for S1 is much smaller than that for S2, indicating that modification of the DBMS transaction manager is more efficient than instrumentation of application source code in terms of running time.

Table 6: Overhead based on TPC-C transaction

	new_order	order-status	delivery
S0	7.980	1.049	5.465
S1	8.946	1.123	6.277
S2	16.186	3.954	14.428
OH1	12%	7%	15%
OH2	102%	277%	264%

In the TPC-C application, there are 34 queries. We modified the application so that each single query is considered as a transaction, and the data and control flows of the original program are kept unchanged. Then we found the total number of schedules consisting of 3 transactions for the following 4 different situations, and their results are given in Table 7.

W1: any 3 transactions.

W2: 2 of the 3 transactions in the same application instance access the same data element or host variable.

W3: 2 of the 3 transactions in the same application instance are generated from XactPair.

W4: 2 of the 3 transactions in the same application instance are generated from XactPair; only those schedules are considered which match the patterns that may reveal concurrency errors in table 4 or table 5.

Table 7: Schedules analysis

Situation	W1	W2	W3	W4
Schedules	39304	916	127	11

As we can see from Table 7, by taking account of the information about the data element, the number of schedules to consider can be reduced significantly (W2/W1= 2.33%). Similarly, if we also consider data flow information, we can further reduce the number of schedules (W3/W2 = 13.43%). Finally, our proposed method also considers the access patterns; the number of interesting schedules is the smallest (W4/W3 = 8.66%).

7. Related work

Many static and dynamic analysis techniques have been proposed for the difficult problem of detecting concurrency related faults. These techniques must be tailored to the concurrency model of the programming language under test.

Model checking tools [13] systematically explore the state spaces of concurrent/reactive software systems. They have been shown to be effective in verifying many types of properties, including absence of specific concurrency faults. Although there has been substantial progress in applying model checking to software [7], it has not been applied to database application programs, where the very large state space due to the database poses a significant challenge.

Instrumentation techniques are widely used in white-box testing. Contest [9] is a tool for detecting synchronization faults in multithreaded Java programs. The program under test is instrumented with sleep(), yield(), and priority() primitives at points of shared memory accesses and synchronization events. At run time, based on random or coverage decisions, Contest can determine whether the seeded primitive is to be executed. A replay algorithm facilitates debugging by saving the orders of shared memory accesses and synchronization events. This kind of instrumentation technique is integrated into our tools for testing database applications.

Race conditions are the major concurrency error which occurs in concurrent systems. ExitBlock [3] is a practical testing algorithm that systematically and deterministically finds program errors resulting from unintended timing dependencies. ExitBlock executes a program or a portion of a program on a given input multiple times, enumerating meaningful schedules in order to cover many program behaviors. However, in database applications, a race condition occurs when two or more application instances try to access the same data element in the database. This can be controlled by the transaction manager of the DBMS system. A major concern in database applications is how to group queries into transactions so that a balance between efficiency and robustness can be achieved.

Flow analysis techniques are used in many fields such as compilation, code analysis, and reverse engineering. Reachability testing [14] systematically executes all possible orders of operations on shared variables in a program consisting of multiple processes. It parallelizes the executions of different schedules to speed up testing. Reachability testing requires explicit declarations of which variables are shared. In contrast, for DB application testing, we can identify the shared variables (data elements) relatively easily and use this information to limit testing to those schedules that are potentially problematic.

8. Conclusions and future work

To test database applications, we have proposed a framework and have designed and implemented a tool set

to partially automate the testing process. In this paper, we extend our previous work to test transaction concurrency at the isolation level of SERIALIZABLE and transaction atomicity/durability. We have identified the potential offline concurrency problems in database applications. A data flow analysis technique is used to identify interesting and legal schedules, which may reveal concurrency errors. Two approaches are suggested to execute a given schedule. The instrumentation method is also used to test atomicity/durability. Preliminary empirical evaluation based on the TPC-C benchmark is presented and demonstrates our approach's effectiveness and efficiency.

Testing database applications involves testing that transactions lead to consistent database states and transform the database space in manner consistent with the application's specification and that no failures occur in concurrent executions that would not occur in serial executions. This paper focuses on the last of those issues. Techniques for checking consistency are discussed in a related paper [8]. Future work includes techniques to expose potential problems associated with "dirty read", "non-repeatable read", and "phantom", for transactions running at isolation levels lower than SERIALIZABLE. We also plan more extensive empirical evaluation.

Acknowledgments

The authors are grateful for the comments and suggestions from anonymous reviewers. We would thank David Chays, Gleb Naumovich, Torsten Suel, and Alex Delis for helpful discussions.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compipler: Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[2] H. Berenson, et al. "A critique of ANSI SQL Isolation Levels" *ACM Special Interest Group on Management of Data Conference*, San Jose, CA USA 1995.

[3] D. Bruening, J. Chapin, "Systematic testing of multithreaded programs", MIT/LCS Technical Memo, LCS-TM 607, April 2000.

[4] D. Chays, P. Frankl, et al. "A Framework for Testing Database Application" *ACM International Symposium on Software Testing and Analysis*, Portland, Oregon, 2000.

[5] D. Chays, Y. Deng, et al. "An AGENDA for Testing Relational Database Application", *Journal of Software Testing, Verification and Reliability*, to appear.

[6] D. Chays, Y. Deng, "Demonstration of AGENDA Tool for Testing Database Applications", *International Conference on Software Engineering*, Portland, Oregon, USA, 2003.

[7] J. Corbett, M. Dwyer, et al. "Bandera: Extracting finite-state models from Java Source Code", *International Conference on Software Engineering*, Limerick, Ireland, 2000.

[8] Y. Deng, D. Chays and P. Frankl, "Testing database transaction consistency", Technical Report, CIS Department, Polytechnic University, 2003.

[9] O. Edelstein, E. Farchi, et al. "Multithreaded Java program test generation", *IBM Systems Journal*, Vol. 41, 2002.

[10] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley & Benjamin Cummings, 2003.

[11] H. Garcia-monlina, J. Ullman, J. Widom, *Database Systems: the Complete Book,* Prentice Hall, 2000.

[12] J. Gray, A. Reuter, *Transaction Processing: Concept and Techniques,* Morgan Kaufmann Publishers Inc., 1993.

[13] G. Holzmann, "The Spin Model Checker", *IEEE Transaction on Software Engineering*, May 1997.

[14] G. Hwang, K. Tai, and T. Huang, "Reachability testing: an approach to testing concurrent software", *Inernational. Journal on Software Engineering and Knowledge Engineering*, Vol. 5, No. 4, 1995, 493-510.

[15] J. Melton, A. Simon, *SQL: 1999 Understanding Relational language components,* Morgan Kaufmann Publishers Inc., 2002.

[16] P. O'Neil, *Database: Principles, Programming, and Performance*, Morgan Kaufmann Publishers, 2000.

[17] T. Ostrand and M. Balcer. "The category-partition method for specifying and generating functional tests". *Communication of ACM*, Vol. 31 no. 6, 1988.

[18] PostgreSQL Global Development Team, PostgreSQL, http://www.postgresql.org, 2002.

[19] R. Stevens, *Advanced programming in the UNIX environment*, Addison-Wesley, 1993.

[20] TPC-Benchmark C. Transaction Processing Council. http://www.tpc.org, 2002.