# Testing Database-Centric Applications For Causes of Database Deadlocks

Mark Grechanik, B. M. Mainul Hossain, Ugo Buy
University of Illinois at Chicago
Chicago, IL 60607
{drmark,bhossa2,buy}@uic.edu

***Abstract*—Many organizations deploy applications that use databases by sending *Structured Query Language (SQL)* statements to them and obtaining data that result from executions of these statements. Since applications often share the same databases concurrently, database deadlocks routinely occur in these databases. Testing applications to determine how they cause database deadlocks is important as part of ensuring correctness, reliability, and performance of these applications. Unfortunately, it is very difficult to reproduce database deadlocks, since it involves different factors such as the precise interleavings in executing SQL statements.**

**We created a novel approach for *Systematic TEsting in Presence of DAtabase Deadlocks (STEPDAD)* that enables testers to instantiate database deadlocks in applications with a high level of automation and frequency. We implemented STEPDAD and experimented with three applications. On average, STEPDAD detected a number of database deadlocks exceeding the deadlocks obtained with the baseline approach by more than an order of magnitude. In some cases, STEPDAD reproduced a database deadlock after running an application only twice, while no database deadlocks could be obtained after ten runs using the baseline approach.**

## I. Introduction

Many organizations and companies deploy *Database-centric applications (DCAs)*, which use databases by sending *transactions* to them – atomic units of work that contain *Structured Query Language (SQL)* statements [14] – and obtaining data that result from the execution of these SQL statements. When DCAs use the same database at the same time, concurrency errors are observed frequently, and these errors are known as *database deadlocks*, which is one of the reasons for major performance degradation in these applications [24] [16, pages 163,223]. The responsibility of relational database engines is to provide layers of abstractions to guarantee *Atomicity, Consistency, Isolation, and Durability (ACID)* properties [14]; however, these guarantees do not include freedom from database deadlocks.

In general, deadlocks occur when two or more threads of execution lock some resources and wait on other resources in a circular chain, i.e., in a *hold-and-wait cycle* [7]. Database deadlocks occur within database engines and not within DCAs that use these databases. A condition for observing database deadlocks is that a database should simultaneously service two or more transactions that come from one or more DCAs, and these transactions contain SQL statements that share the same resources (e.g., tables or rows). In enterprise systems, database deadlocks may appear when a new transaction is issued by a DCA to a database that is already used by some other legacy DCA, thus making the process of software evolution error-prone, expensive, and difficult.

Currently, database deadlocks are typically detected within database engines using special algorithms that analyze whether transactions hold resources in cyclic dependencies, and these database engines resolve database deadlocks by forcibly breaking the hold-and-wait cycle [1], [14], [16]–[18], [23], [28]. That is, once a deadlock occurs, the database rolls back one or more of the transactions that is involved in the circular wait. Doing so effectively resolves the database deadlock, however, exceptions are thrown in the components of the DCAs that sent these aborted transactions.

Essentially, these exceptions are notification mechanism to let stakeholders know that a business flow is disrupted, since some transactions did not go through. Next, stakeholders study causes of database deadlocks, so that they can avoid them. Programmers are advised to practise defensive programming by writing special database deadlock exception-handling code, for example, to repeat aborted transactions when applicable – searching for “database deadlock exception” on the Web yields close to 2,500 web pages, many of which instruct programmers on how to handle database deadlock exceptions for different databases. However, this solution is considered a temporary patch, since it leads to significant performance degradation – detecting a database deadlock, throwing an exception, and retrying a transaction comes at a high cost. Thus, it is important to analyze these exceptions to determine the causes of database deadlocks, so that they can be avoided altogether by refactoring DCAs[1].

Different database deadlock avoidance programming patterns help database designers and programmers structure their code, transactions, and data so that they can avoid database deadlocks [11], [15], [22] [27, pages 249–252]. For example, Microsoft published guidelines for minimizing database deadlocks in SQL Server[2]. These guidelines include, among others, accessing database objects in the same order, avoiding user interaction in transactions, and keeping transactions short and in one batch. Since these solutions are manual and error-prone,

[1]http://stackoverflow.com/questions/595187/orm-support-for-handling-deadlocks. Last verified December 22, 2012.

[2]http://msdn.microsoft.com/en-us/library/ms191242(v=sql.105).aspx. Last verified on December 22, 2012.

it is important to reproduce database deadlocks using DCAs automatically during testing to see what avoidance patterns fit best. Consistently and systematically reproducing database deadlocks is very difficult and laborious, since identifying execution scenarios that lead to database deadlocks requires sophisticated reasoning about the combined behavior of DCAs and their databases. The result of this process is overwhelming complexity and a significant cost of reproducing database deadlocks. Our interviews with different Fortune 100 companies confirmed that database deadlocks occur on average every two to three weeks for large-scale enterprise DCAs, some of which have been around for over 20 years. For instance, database deadlocks still occur every ten days on average in a commercial large-scale DCA that handles over 70% of cargo flight reservations in the USA. In this case, a test engineer would wait for ten days in order to detect a single database deadlock, which is obviously impractical.

We created a novel approach for *Systematic TEsting in Presence of DAtabase Deadlocks (STEPDAD)* that enables testers to instantiate database deadlocks in DCAs with a high level of automation and frequency. This paper makes the following contributions.

- We model transactions using lock graphs to detect hold-and-wait cycles in transactions.
- STEPDAD exploits information about hold-and-wait cycles to explore interleavings of queries performed by different transactions using a technique called execution hijacking [32]. These interleavings attempt to produce deadlocks matching the detected hold-and-wait cycles, thereby significantly increasing the probability of database deadlocks.
- We implemented STEPDAD and experimented using three client/server DCAs. On average STEPDAD produced a number of database deadlocks exceeding by more than an order of magnitude the number of database deadlocks obtained the baseline approach. In some cases, STEPDAD produced a database deadlock after running an application only two times, while no database deadlocks were produced after ten runs using the baseline approach. Our tool is publically available at http://www.cs.uic.edu/~drmark/STEPDAD.htm.

## II. The Problem

In this section we give an illustrative example of a database deadlock, show how DCAs use databases, and formulate the problem statement.

### A. An Illustrative Example

Consider the example of database deadlock shown in Table I. Transactions $T_1$ and $T_2$ are independently sent by DCAs to the same database at the same time. When the first DCA executes the `UPDATE` statement in Step 1, the database locks rows of table `authors` in which the value of attribute `paperid` is 1. Next, the second DCA executes the `UPDATE` statement in Step 2 and the database locks rows of table `titles` in which attribute `titleid` is 2. When the `SELECT`

TABLE I
Example of a database deadlock that may occur when two transactions $T_1$ and $T_2$ are issued by DCA(s).

| **Step** | **Transaction $T_1$** | **Transaction $T_2$** |
|---|---|---|
| 1 | UPDATE authors SET citations=100 WHERE paperid=1 | |
| 2 | | UPDATE titles SET copyright=1 WHERE titleid=2 |
| 3 | SELECT title, doi FROM titles WHERE titleid=2 | |
| 4 | | SELECT authorname FROM authors WHERE paperid=1 |

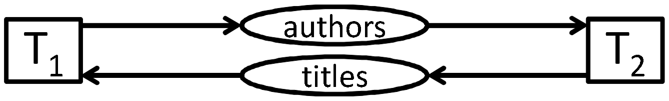


Fig. 1. A lock graph for the transactions shown in Table I. The lock graph shows the hold-and-wait cycle $T_1 \rightarrow$`authors`$\rightarrow T_2 \rightarrow$`titles`$\rightarrow T_1$.

statement in Step 3 is executed as part of transaction $T_1$, the database attempts to obtain a read lock on the rows of table `titles`, which are exclusively locked by transaction $T_2$ of the second DCA.

Since these locks cannot be imposed simultaneously on the same resource (i.e., these locks are not compatible), $T_1$ is put on hold. Finally, the `SELECT` statement in Step 4 is executed as part of transaction $T_2$; the database attempts to obtain a read lock on the rows of table `authors`, which are exclusively locked by transaction $T_1$ of the first DCA. At this point both $T_1$ and $T_2$ are put on hold resulting in a database deadlock.

Figure 1 shows the lock graph for the transactions appearing in Table I. Transactions are depicted as rectangles and resources (i.e., tables) are shown as ovals. Arrows directed towards resources designate locks held by transactions on those resources; arrows in the opposite direction designate transactions that are waiting to obtain resource locks. The lock graph shows the hold-and-wait cycle $T_1 \rightarrow$`authors`$\rightarrow$ $T_2 \rightarrow$`titles`$\rightarrow T_1$. This hold-and-wait cycle may not always result in a database deadlock; however, when interleaving steps occur as shown in Table I, a database deadlock is highly likely. One exception is when these tables contain little or no data; in this case, locks may be released by the database engine almost instantaneously or not imposed at all.

### B. How DCAs Use Databases

Many enterprise-level DCAs are written in general-purpose programming languages (e.g., Java); they communicate with relational databases by using standardized *application programming interfaces (APIs)*, such as *Java DataBase Connectivity (JDBC)*. Using JDBC, programs pass SQL statements as string parameters in API calls that send these SQL statements to databases for execution. For example, the API call `executeQuery` of the class `Statement` takes a string containing an SQL statement that is sent to a database for execution. Once executed, values of database attributes that are specified in SQL statements are returned to DCAs using JDBC's `ResultSet` interface. These SQL statements are executed as part of a transaction that is delimited by statements "begin transaction" (by setting the connection's autocommit mode to `false`) and "end transaction" with the subsequent API call `commit`. In case a transaction is not explicitly delimited in the source code, each SQL statement is taken to be a separate transaction, which may be committed automatically by the database.

We observe that large-scale multi-tiered software applications have significantly more complex interactions with databases through a variety of components that are organized in different tiers. For example, an *Enterprise Claim Application (ECA)* at a major insurance company integrates different databases and programming components (some of which are legacy components), which include database *triggers* and *stored procedures*, which are database objects that consist of SQL statements and some fourth-generation language statements designed to work with SQL. Essentially, a stored procedure is a function written in a high-level language that resides within the database. Database triggers are stored procedures associated with certain operations on database objects (e.g., tables and rows) [6], [8], [30]. Triggers autonomously react to database events by evaluating a data-dependent condition and by executing a reaction whenever the condition is satisfied [21]. In addition, engineers develop database plugins that are programming components written in general-purpose programming languages and invoked in response to events that occur within database engines [13]. Stored procedures, triggers, and plugins may be written by different programmers, and it is difficult to determine the execution path that leads to database deadlocks.

In the ECA, which is a representative of many large-scale applications, a transaction sent by some component to a database invokes an internal program that sets off a chain of invocations. For example, this internal program can be a trigger that calls an external component that sends a new transaction to a different database that invokes a stored procedure that executes a different transaction that results in a database deadlock. In this kind of situation, it is not only important to identify what components catch database deadlock exceptions, but also to determine the execution trace that leads to the deadlock. Knowing a precise set of invocations that leads to a database deadlock enables stakeholders to design a strategy to avoid this deadlock.

## C. Reproducing Versus Simulating Database Deadlocks

Reproducing database deadlocks is difficult, since it involves executing applications using certain input data that lead to sending specific transactions to databases that have hold-and-wait cycles that can be realized as database deadlocks. An alternative approach is to simulate database deadlocks using *mock objects* that throw exceptions when a transaction is sent to the database. That is, a mock object represents databases, making it easy for programmers to test their exception-handling code without having actual databases. While this idea offers a simple implementation and may be effective in a number of situations, there are drawbacks. In our ECA example, different exception objects propagate through layers of software by being caught and sometimes re-thrown. The logic of exception handling depends on the application, and it is often unclear for testers how an exception should be handled, if at all. Understanding how database deadlocks are caused is equally or more important.

In addition, database deadlock resolution and exception throwing mechanisms are not perfect. In some cases, database deadlocks are incorrectly processed by database layers leading to null pointer exceptions[3]. Mock objects offer very limited benefits in such cases, because the source of the problem can be traced only with the actual database deadlocks. In certain cases, retrying aborted transactions is not an acceptable solution, since it may interfere with timing constraints on the application (e.g., high-frequency stock trading) and lead to an incorrect state of the database. This is why it is often more valuable to actually reproduce database deadlocks rather than just simulating them.

The other drawback is that using mock objects does not allow testers to observe actual database deadlocks and to obtain from the database engine SQL execution traces showing how database deadlocks happen. Database deadlocks often involve complex interactions among different database objects, with hold-and-wait cycles among transactions. SQL Server has over dozen of documented kinds of database deadlocks[4] besides the classic one that we showed using the illustrative example in Table I. For example, there is a database deadlock that occurs during *savepoint rollback*—a complicated set of locks are imposed by objects of the database engine leading to a database deadlock even if there are no hold-and-wait cycles in SQL statements of the transactions that are concurrently issued by different DCAs to the same database. Thus, it is important to produce database deadlocks, so that database administrators, testers, and developers can understand the runtime concurrent behavior of their DCAs and ways to improve it.

## D. The Problem Statement

This paper addresses the following main question: How does one test existing DCAs that share the same databases for causes of database deadlocks arising from transactions issued by these DCAs? Therefore, our main goal is to reproduce database deadlocks with a high level of automation and frequency, thus enabling software engineers to determine how these database deadlocks are caused. It is not a goal of our solution to reproduce all database deadlocks that involve all potential hold-and-wait cycles among different transactions, but instead to reproduce as many database deadlocks as possible.

[3]http://www-01.ibm.com/support/docview.wss?uid=swg1JR33072. Last verified on December 22, 2012.

[4]http://sqlindian.com/2012/07/06/sql-server-deadlocks-and-live-lockscommon-patterns. Last verified on December 22, 2012.

If hold-and-wait cycles are detected statically in different transactions, the information about these cycles can be used to force these transactions to execute simultaneously by the database engine, thus increasing the probability of occurrence of database deadlocks. Once statically detected, these hold-and-wait cycles may potentially result in database deadlocks. However, depending on interleavings in different execution scenarios, not all of these cycles will lead to database deadlocks, meaning that *false positives (FPs)* are possible. Given that predicting when deadlocks will occur is practically impossible, we would rather err on the side of FPs.

Our approach should not depend on a specific database engine or require modifications of the kernels of database engines. Similar to operating systems, database engines are very complex, fragile, and closed software systems; a solution that requires their modification is unlikely to be practical. Moreover, our approach should not depend on specific architectures of DCAs. Finally, it is important for STEPDAD to be efficient, that is, it should not take an excessive amount of time to reproduce a database deadlock. A baseline method is to run DCAs for some time to observe database deadlocks. Different runs of DCAs yield different times to database deadlocks because of their probabilistic nature. The values of *mean time to database deadlock (MTTDD)* should be much smaller with STEPDAD when compared with the baseline approach.

## III. Our Approach

In this section we describe our key ideas and the abstraction on which they are based, we explain the architecture of STEPDAD, we describe the cycle detection algorithm that we use to detect hold-and-wait cycles in lock graphs, and we show how STEPDAD schedules transactions to increase the probability of database deadlock occurences.

### A. Our Abstraction

STEPDAD is based on our abstraction that represents relational databases as sets of resources (e.g., database tables) and transactions that DCAs issue to databases as sets of abstract operations. Examples of abstract operations include reading from and writing into resources; these operations also issue synchronization requests. Using this abstraction unifies DCAs that use the same databases in a novel way: their independently issued transactions become abstract operations with resource sharing requests. With this abstraction, we hide the complex machinery of database engines. Instead we focus on abstract operations performed by transactions and the engines' concurrency control locking mechanisms associated with these abstract operations.

### B. Key Ideas

Our solution rests on a key idea that transactions involved hold-and-wait cycles should be executed simultaneously in order to increase the probability that a deadlock will occur. We specifically introduce a mechanism for scheduling executions of DCAs in such a way that these transactions will be issued in close temporal proximity. A rationale for this idea is that database schedulers are more likely to create interleavings of instructions that realize hold-and-wait cycles if transactions arrive at the same time. This idea is related to work on producing scheduling that causes concurrent programs to fail [3]. Of course, there is no guarantee that the simultaneous arrival of transactions may result in a database deadlock—this is a hypothesis that we evaluate in Section V.

The other key idea is to replicate transactions that have hold-and-wait cycles by issuing them simultaneously from different client applications to further increase the probability of observing database deadlocks. Consider the motivating example that is shown in Table I. By adding two or more replicated transactions $T_1$ and $T_2$ we can increase the "density" of SQL statements that contain hold-and-wait cycles per time unit of processing. Our hypothesis is that by increasing the number of replicated transactions that contain hold-and-wait cycles and that are sent to the database simultaneously, we can increase the probability of occurrence of database deadlocks. We evaluate this hypothesis and report the results of our evaluation in Section V.

### C. The Architecture of STEPDAD

The design of STEPDAD rests on a common set of requirements to realize our key ideas. The first requirement is that a DCA should be forced to issue transactions to databases for execution. The next requirement is to intercept all sent transactions either to simulate exceptions thrown by the database or to cause a delay long enough to accumulate a set of different transactions that form hold-and-wait cycles. Finally, the third requirement is to increase the "density" of transactions to force a database deadlock faster. These requirements define the architecture of STEPDAD that is shown in Figure 2. This architecture shows two DCAs (i.e., DCA$_m$ and DCA$_n$) that use the shared Database as it is shown with dashed arrows labeled with (1). We also assume that stored procedures, triggers, and other database objects that contain transactions are incorporated in the Transactions block in this architecture.

The first step in this architecture involves extracting transactions that contain SQL statements from these DCAs as shown with dashed arrows labeled with (2). In addition, SQL statements from other transactions from database objects can be extracted. This is a one-time manual effort that may be required (as it was done for three subject DCAs in this paper). At first glance, it appears to be tedious and laborious work for programmers to extract SQL statement from the source code of DCAs. In reality, it is a practical and modest exercise that takes little time. We observed in industry that all transactions with SQL statements are available in separate documents for many enterprise applications. The explanation is simple—transactions contain complicated SQL statements that should be debugged and tested by database analysts using specialized SQL development environments before they are used by developers of DCAs.

Once transactions are extracted, the static analysis phase starts. First, (3) SQL statements that are contained in these

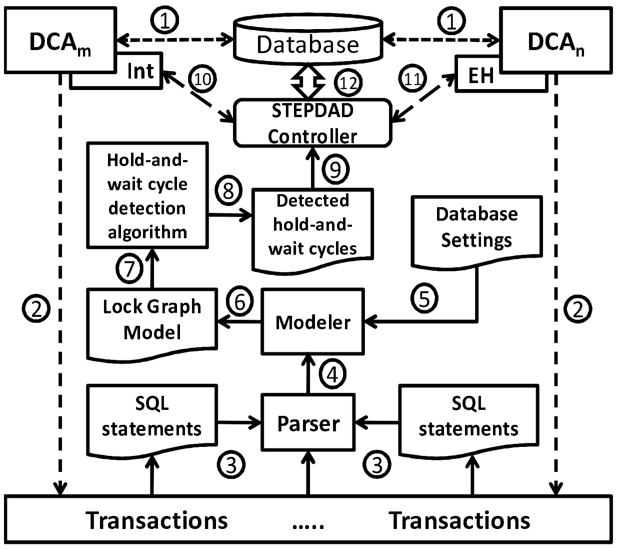


Fig. 2. The architecture of STEPDAD.

transactions are parsed, (4) and the resulting parse trees are input into the Modeler that automatically transforms SQL statements into the abstract operations and synchronization requests. In STEPDAD, we extracted the SQL parser from the Apache Derby database. The Modeler (5) uses database settings that include a locking strategy (6) to produce a lock-graph model, similar to the one that is shown in Section II-A. This model (7) serves as the input to an algorithm that (8) detects at most one hold-and-wait cycle between each pair of transactions, that are in turn (9) used as inputs to the STEPDAD Controller. This step concludes the static phase of STEPDAD.

At this point, we describe the dynamic phase during which DCAs are run for the purpose of producing database deadlocks. We accomplish this goal by diverting SQL statements from DCAs to the Controller, which can then determine whether executing these SQL statements may result in the creation of hold-and-wait cycles and therefore deadlocks. STEPDAD diverts SQL statements with the *interceptor pattern* whose implementation uses a framework associating call-backs with particular events [29]. In STEPDAD, we use AspectJ[5] to instrument subject DCAs with aspects to intercept SQL statements, even though different binary rewriting tools can be used for this purpose. We extend this framework by writing modules that register user-defined call-backs with the framework. When a framework event arises, the registered methods are invoked thereby alerting the user to these events.

The first step is to add *interceptors* to DCAs. These interceptors are shown as a partial rectangle labeled `Int` and positioned next to the corresponding DCA in Figure 2. These interceptors trap JDBC API calls that take SQL statements as string parameters from DCA$_m$ and instead of (1) sending these statements to the Database, (10) they divert them to the Controller, whose goal is to quickly look up if hold-and-wait cycles are present that involve SQL statements in other transactions, for example, from DCA$_n$. In doing so, the Controller utilizes the information obtained from the static analysis phase. Once the Controller determines what other SQL statements have hold-and-wait cycles with the pending SQL statements, (11) the Controller sends instructions to the execution hijacking layer (EH) [32] that forces DCA$_n$ to execute the statement containing the JDBC API call with the conflicting SQL statement. Once both SQL statements are pending at the Controller, (12) the Controller forwards these SQL statements to the database for execution, thus increasing the probability that the database deadlock will actually occur. This concludes the description of STEPDAD's architecture.

[5]http://www.eclipse.org/aspectj, verified September 17, 2012.

**Algorithm 1** The cycle detection algorithm for lock graphs.

```
1:  CycleDetect( Lock graph $\mathcal{G}$ )
2:  $\forall t \in T$, color($t$) ← BLUE{Initialize color of each transaction node.}
3:  for all $t \in T$ do
4:    if color($t$) = BLUE then
5:      DFS_visit($t$)
6:    end if
7:  end for
8:  return
9:  DFS_visit( Transaction $v$ )
10: color($v$) ← RED
11: for all $w \in T$, s.t. $(v,r) \in E \wedge (r,w) \in E \wedge r \in R$ do
12:   if color($w$) = RED $\wedge w \rightsquigarrow v$ then
13:     print $w \rightsquigarrow v$
14:   else if color($w$) = BLUE then
15:     DFS_visit($w$)
16:   end if
17: end for
18: color($v$) ← WHITE
```

## D. The Cycle Detection Algorithm

The procedure `CycleDetect` is shown in Algorithm 1 that is a variation of the algorithm for finding cycles in a directed graph using *depth-first search (DFS)* [31]. This algorithm takes as its input the lock graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. The set of nodes, $\mathcal{V}$ is the union of the set of transactions, $T$ and the set of resources, $R$. Directed edges designate read and write locks by connecting transactions and resources. Hold-and-wait cycles are computed and returned in Line `13` of the algorithm. where the arrow $\rightsquigarrow$ designates a hold-and-wait cycle path between transactions $w$ and $v$. The algorithm recursively calls the procedure `DFS_visit` (see Line `15`) that performs a depth-first search on the graph $\mathcal{G}$. A key idea of this algorithm is to exploit the DFS to compute hold-and-wait cycles with respect to transactions that hold locks on resources.

In line 2 of Algorithm 1 all transaction nodes are initialized with the color `BLUE`. Lines 3–7 iteratively check all such nodes. Any transaction node colored BLUE is explored by invoking procedure DFS_visit, which searches recursively for hold-and-wait cycles.

Lines 9–18 in Algorithm 1 specify the body of the procedure DFS_visit. The procedure takes as input the transaction node *v*. Line 10 initializes the color of this transaction node to RED. Lines 11–17 iterate through other transaction nodes that are connected to this node via shared locks on resources. If a transaction node is encountered that is already colored RED, then a hold-and-wait cycle is found and reported in Line 13. Line 18 paints node *v* WHITE meaning that this node does not participate in any cycle and can be ignored. The algorithm terminates when all transaction nodes are explored.

We implemented Algorithm 1 in a hold-and-wait cycle detection tool whose GUIs are shown in Figure 3. On the left, the GUI shows a list of transactions, one per list box with SQL statements that constitute a transaction. On the right, a detected hold-and-wait cycle is visualized as a lock graph.

## IV. Experiments

In this section we describe our empirical evaluation of STEPDAD on three small Java DCAs.

### A. Research Questions

We seek to answer the following research questions.

**RQ1**: Can STEPDAD effectively reproduce database deadlocks for DCAs that issue transactions to the database?

**RQ2**: Is STEPDAD efficient in reproducing database deadlocks?

The rationale for both RQs is to compare STEPDAD with the baseline approach where multiple DCAs are run for a certain period of time until a database deadlock is registered or until a time period expires. Our goal is to show that STEPDAD is more effective and efficient than this baseline approach. We address RQ1 by measuring the number of deadlocks reported by STEPDAD vs. the baseline approach. The rationale for RQ2 is determine the *mean time to database deadlock (MTTDD)*. To address RQ2 we measure STEPDAD's MTTDD and compare it with the baseline approach.

### B. Subject DCAs

We evaluate STEPDAD with three Java DCAs whose characteristics are shown in Table II. HIM is a program for maintaining health information records. DAN is a demographic analysis program. Finally, UCOM is a program for obtaining statistics on how users interact with Unix systems using their commands. These DCAs were created by 32 graduate students (divided into three groups) from the University of Illinois at Chicago who wrote them as part of a graduate course of distributed object programming using simplified specifications from real-world applications that came from different projects at Accenture. Databases were created using UCI Knowledge Discovery in Databases Archive[6]. The subject DCAs as well

TABLE II
Subject DCAs and their databases. The columns show lines of code (LOC) in DCAs, the size of DB in megabytes, the number of transactions, $T$ in the DCA and how many SQL statements, $S$ at most are contained in each transaction, the number of tables in the database, $T_{DB}$, the number of tables used in transactions, $T_{trans}$, and the total number of rows in the database.

| App | LOC | DB Size | $T$ | $S$ | $T_{DB}$ | $T_{trans}$ | Rows |
|---|---|---|---|---|---|---|---|
| HIM | 3,421 | 248MB | 4 | 2 | 10 | 6 | 1,330,107 |
| UCOM | 2,127 | 29MB | 2 | 2 | 8 | 2 | 250,532 |
| DAN | 6,034 | 371MB | 2 | 2 | 13 | 2 | 1,270,897 |

as their databases are available from Sourceforge[7]. Each DCA consists of the server component that spawns multiple threads that use its backend database, and a client component that submits client requests and obtains data from the server.

### C. Methodology

We aligned our methodology with the guidelines for statistical tests to assess randomized approaches in software engineering [2]. Since database deadlocks are not easy to reproduce, different runs of the DCA may reveal different number of deadlocks and different MTTDDs. Our goal is to collect highly representative samples of runs when applying different approaches, perform statistical tests on these samples, and draw conclusions from these tests. Since our experiments involve the probability of encountering database deadlocks, it is important to conduct the experiments multiple times to pick the average to avoid skewed results. For each subject DCA, we ran each experiment 10 times with each approach to obtain a good representative sample.

*1) Independent Variables:* We have three independent variables: the subject DCA, the type of the experiment, and the number of DCA clients. There are three types of experiment: the **R**egular or baseline, where a subject DCA is run without STEPDAD, the experiment with the **C**ontroller that enables simultaneous delivery of transactions from subject DCA clients to the database for execution, and finally, the experiment with **A**rtificially replicated transactions to increase concurrency. For each subject DCA, we carried out experiments with two, four, and six clients for five minutes per experiment.

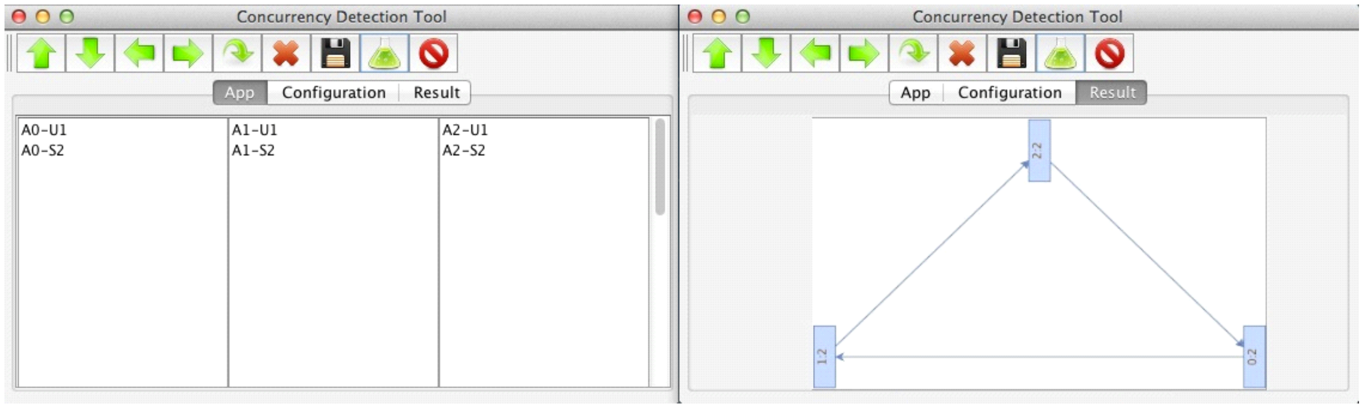
*2) Dependent Variables:* We have two dependent variables: the number of database deadlocks that are observed during the experiment and the running time of each subject DCA until at least one database deadlock is encountered. Since exhibiting database deadlocks requires specific interleavings of transactions, we repeated each experiment 10 times. Thus, the total number of experiments is equal to three DCAs $\times$ three types (R,C,A) $\times$ three client settings $\times$ 10 times = 270 experiments. We report statistical results (average, median, min, max, variance) for 10 runs for the number of observed deadlocks and the time taken to run the subject DCAs into these deadlocks.

### D. Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for MTTDD for different

[6]http://kdd.ics.uci.edu, last verified on August 16, 2012.

[7]http://sourceforge.net/projects/redactapps, verified on September 17, 2012.



```
A0-U1        A1-U1        A2-U1
A0-S2        A1-S2        A2-S2
```

Fig. 3. The GUI of STEPDAD's hold-and-wait cycle detection tool. On the left, the GUI shows a list of transactions, one per list box with SQL statements that constitute a transaction. On the right, a detected hold-and-wait cycle is visualized as a lock graph.

approaches. We seek to evaluate the following hypotheses at a 0.05 level of significance.

- $H_0$ The primary null hypothesis is that there is no difference in the values of MTTDD between R, C, and A approaches for all subject DCAs.
- $H_1$ An alternative hypothesis to $H_0$ is that there is statistically significant difference in the values of MTTDD between R, C, and A approaches for all subject DCAs.

Once we test the null hypothesis $H_0$, we are interested in the directionality of means, $\mu$, of the results of control and treatment groups. We are interested to compare the effectiveness of STEPDAD versus the R and A approaches.

- H1 (MTTDD of R versus C). The effective null hypothesis is that $\mu_R = \mu_C$, while the true null hypothesis is that $\mu_R \leq \mu_C$. Conversely, the alternative hypothesis is $\mu_R > \mu_C$.
- H2 (MTTDD of R versus A). The effective null hypothesis is that $\mu_R = \mu_A$, while the true null hypothesis is that $\mu_R \leq \mu_A$. Conversely, the alternative hypothesis is $\mu_R > \mu_A$.
- H3 (MTTDD of C versus A). The effective null hypothesis is that $\mu_C = \mu_A$, while the true null hypothesis is that $\mu_C \leq \mu_A$. Conversely, the alternative is $\mu_C > \mu_A$.

The rationale behind the alternative hypotheses to H1–H3 is that STEPDAD allows testers to quickly reproduce database deadlocks. These alternative hypotheses are motivated by our belief that by reducing the number of interleavings among transactions that contain hold-and-wait cycles in addition to increasing concurrency by replicating these transactions, they can result in reproducing database deadlocks much faster when compared to the baseline approach.

### E. Threats to Validity

A threat to the validity of this experimental evaluation is that our subject programs are relatively small; it is difficult to find large open-source DCAs that use nontrivial databases. Large DCAs may have millions of lines of code and use databases whose sizes are measured in thousands of tables and attributes. Those DCAs and databases may have different characteristics compared to our smaller subject programs. On the one hand, increasing the size of applications to millions of lines of code is unlikely to affect the time and space demands of our analyses because STEPDAD only considers transactions. Thus, the source code of DCAs is ignored in the cycle analysis, which is focused on the transactions that these DCAs issue to their databases. On the other hand, increasing the size and the number of transactions may have a significant impact on the cost of cycle analysis. In addition, it may be more challenging to schedule the execution of large and complex applications to reproduce database deadlocks. Evaluating this impact is a subject of future work.

Additional threats to validity of this study is that we used graduate students as programmers who created DCAs, and this task should be tackled by professional programmers. However, most of these students have at least one year of professional programming experience, thereby reducing this threat to validity.

Finally, recall that there are over two dozens of different kinds of database deadlocks. In this paper, we experimented only with circular database deadlocks that are realized from hold-and-wait cyclic locks on resources by transactions. It is unclear how well STEPDAD will perform on other kinds of database deadlocks, so this is a threat to external validity of our results.

## V. Results

In this section, we report the results of our experiment and evaluate the hypotheses. We use one-way ANOVA and t-tests for paired two sample for means to evaluate the hypotheses that we stated in Section IV.

### A. Analyzing Experimental Results

Experimental results are summarized and shown in Table III. We use dash instead of reporting time for the experiments where no database deadlocks were reproduced. For DCAs HIM and UCOM with two clients for the regular baseline experiments, database deadlocks were not reproduced at all. Even for the subject DCA HIM with six concurrent

TABLE III

Each subject DCA (i.e., HIM, UCOM and DAN) is evaluated using (**R**)egular, (**C**)ontroller and (**A**)rtificial injection methodologies. Each DCA was run with 2, 4 and 6 clients for 5 mins per experiment. We measured the number of detected deadlocks in 5 mins of the experiment(in column deadlocks) and also the time for the detection of the first deadlock within 5 mins (in column TIME). For the columns Deadlock and Time, we report statistical results (MTTDD, median, min, max, variance) of 10 runs for each DCA/client setting. We use dash instead of reporting time for the experiments where no database deadlocks were reproduced.

| DCA | | | Deadlocks | | | | | Time (in seconds) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | *Type* | *Clients* | *Avg* | *Med* | *Min* | *Max* | *Var* | *MTTDD* | *Med* | *Min* | *Max* | *Var* |
| HIM | R | 2 | 0 | 0 | 0 | 0 | 0 | - | - | - | - | - |
| | | 4 | 0.6 | 0.5 | 0 | 2 | 0.49 | 146.4 | 102 | 56 | 262 | 9883.3 |
| | | 6 | 1.2 | 1 | 0 | 3 | 1.73 | 169.2 | 197 | 86 | 240 | 4917.7 |
| | C | 2 | 0.6 | 0.5 | 0 | 2 | 0.49 | 169.8 | 171 | 120 | 209 | 1610.7 |
| | | 4 | 0.9 | 1 | 0 | 2 | 0.32 | 161 | 148.5 | 65 | 256 | 5128.57 |
| | | 6 | 3 | 3 | 2 | 4 | 0.44 | 80.1 | 74.5 | 58 | 122 | 433.88 |
| | A | 2 | 1.5 | 1.5 | 0 | 3 | 1.17 | 129.38 | 115 | 53 | 270 | 5784.27 |
| | | 4 | 2.5 | 2.5 | 0 | 5 | 2.5 | 103.44 | 70 | 59 | 266 | 4576.53 |
| | | 6 | 4.5 | 4.5 | 3 | 6 | 1.61 | 95.1 | 87.5 | 46 | 194 | 1838.1 |
| UCOM | R | 2 | 0 | 0 | 0 | 0 | 0 | - | - | - | - | - |
| | | 4 | 0.2 | 0 | 0 | 1 | 0.18 | 46.5 | 46.5 | 39 | 54 | 112.5 |
| | | 6 | 7.1 | 7 | 3 | 13 | 9.66 | 57.5 | 47.5 | 32 | 146 | 1132.28 |
| | C | 2 | 0 | 0 | 0 | 0 | 0 | - | - | - | - | - |
| | | 4 | 1.3 | 1 | 0 | 3 | 0.9 | 115.88 | 109 | 36 | 226 | 3965.84 |
| | | 6 | 7.4 | 7.5 | 2 | 12 | 10.93 | 76.6 | 61.5 | 22 | 207 | 3663.38 |
| | A | 2 | 1.5 | 1.5 | 0 | 3 | 0.72 | 102.11 | 108 | 48 | 149 | 1228.11 |
| | | 4 | 10.9 | 10 | 6 | 15 | 7.43 | 48.6 | 47.5 | 23 | 72 | 301.6 |
| | | 6 | 22.3 | 22 | 15 | 30 | 15.34 | 32.2 | 31.5 | 22 | 48 | 59.73 |
| DAN | R | 2 | 0.9 | 1 | 0 | 2 | 0.54 | 173.86 | 206 | 69 | 264 | 6056.48 |
| | | 4 | 5 | 5.5 | 1 | 9 | 5.56 | 85.3 | 86 | 32 | 147 | 1318.01 |
| | | 6 | 10.5 | 10.5 | 7 | 15 | 5.83 | 51.6 | 40 | 26 | 138 | 1326.71 |
| | C | 2 | 1.9 | 2 | 1 | 3 | 0.77 | 136.8 | 123.5 | 56 | 251 | 4815.96 |
| | | 4 | 6.7 | 6.5 | 4 | 10 | 4.68 | 54.7 | 48 | 36 | 107 | 436.68 |
| | | 6 | 9.7 | 10 | 5 | 13 | 5.57 | 47.2 | 38.5 | 31 | 85 | 325.96 |
| | A | 2 | 3.7 | 4 | 2 | 5 | 1.12 | 83.5 | 69 | 36 | 188 | 2749.83 |
| | | 4 | 6.8 | 7 | 4 | 8 | 1.96 | 54.8 | 50 | 35 | 112 | 571.73 |
| | | 6 | 11.3 | 11.5 | 7 | 16 | 7.34 | 52.3 | 41.5 | 31 | 103 | 639.57 |

clients database deadlocks were not reproduced in five out of ten regular baseline experiments. In contrast, with the artificial injection experiment, database deadlocks were reproduced in almost all experiments, except for two experiments with two clients for HIM, one experiment with four clients for HIM, and one experiment for two clients for UCOM.

In total, for the regular baseline approach, 20 out of 30 experiments did not result in any database deadlocks for HIM; 18 out of 30 experiments did not result in any database deadlocks for UCOM; and three out of 30 experiments did not result in any database deadlocks for DAN. For the Controller-based approach, only seven out of 30 experiments did not result in any database deadlocks for HIM; 12 out of 30 experiments did not result in any database deadlocks for UCOM; and all 30 experiments resulted in database deadlocks for DAN. Finally, for the artificially injected transactions approach, only three out of 30 experiments did not result in any database deadlocks for HIM; only one out of 30 experiments did not result in any database deadlocks for UCOM; and all 30 experiments resulted in database deadlocks for DAN. Based on these results, we can positively answer RQ1 that STEPDAD effectively reproduces at least one database deadlock.

## B. Testing the Null Hypothesis

We used ANOVA to evaluate the null hypothesis $H_0$ that the variation in an experiment is no greater than that due to normal variation of DCAs' characteristics and measurement errors given the probabilistic nature of reproducing database deadlocks. The results of ANOVA confirm that there are large differences between the groups in terms of MTTDD for HIM for two clients with $F = 5.64 > F_{crit} = 3.35$ with $p \approx 0.009$ which is strongly statistically significant. Similarly, the results of ANOVA for MTTDD for UCOM for two clients with $F = 35.6 > F_{crit} = 3.35$ with $p \approx 9.4 \cdot 10^{-9}$ which is strongly statistically significant. However, the results of ANOVA for MTTDD for DAN for two clients are inconclusive with $F = 1.2 < F_{crit} = 3.35$ with $p \approx 0.31$, while for four clients the differences are statistically significant with $F = 4 > F_{crit} = 3.35$ with $p \approx 0.03$. Based on these results we reject the null hypothesis and we accept the alternative hypothesis $H_1$.

## C. Comparing Baseline with Controller

To test the null hypothesis `H1`, we applied two t-tests for two paired sample means, in this case MTTDDs for the regular baseline and the controller approaches. Since we did not reproduce database deadlocks with the regular baseline approach for a number of experiments, we cannot run t-tests on these results. Instead, we statistically evaluated UCOM for six clients, and DAN for four and six clients. Statistical evaluation of UCOM is inconclusive, and for DAN for four clients, $t > t_{crit}$ as $2.25 > 1.85$ with $p \approx 0.025$ and for six clients it is inconclusive with $MTTDD_R = 51.6 > MTTDD_C = 47.2$. Based on these results we reject the null hypothesis for a general case `H1` and we accept the alternative hypothesis that states that **MTTDD for STEPDAD with Controller is generally lower than MTTDD for the regular baseline approach**.

### D. Comparing Baseline with Artificial

To test the null hypothesis H2, we applied two t-tests for two paired sample means, in this case MTTDDs for the regular baseline and the artificial injection approaches. Since we did not reproduce database deadlocks with the regular baseline approach for a number of experiments, we cannot run t-tests on these results. Instead, we statistically evaluated UCOM for six clients, and DAN for four and six clients. Statistical evaluation of UCOM show that the artificial approach results in smaller MTTDD with $MTTDD_R = 57.5 > MTTDD_C = 32.2$ with $t > t_{crit}$ as $2.27 > 1.83$ with $p \approx 0.027$. For DAN for four clients, the result is strongly statistically significant and for six clients it is inconclusive. Based on these results we reject the null hypothesis for a general case H2 and we accept the alternative hypothesis that states that **MTTDD for STEPDAD with Artificial injector is generally lower than MTTDD for the regular baseline approach**.

### E. Comparing Controller with Artificial

To test the null hypothesis H3, we applied two t-tests for two paired sample means, in this case *MTTDD*s for the artificial injector and the controller approaches. Based on these results we accept the null hypothesis H3 that say that **MTTDD for STEPDAD with Artificial injector is generally the same as MTTDD for the controller-based approach**.

### F. Discussion

One important conclusion from our experiments is that replication of database deadlocks is as effective with scheduling transactions using the controller as with artificial injection. Essentially, this is not surprising, since artificially injecting multiple transactions that hold-and-wait cycles is likely to result in a higher frequency of database deadlocks—more contention is created. Of course, database engine is a complex mechanism that parses SQL statements in transactions, translates them into low-level relational algebra operators, and creates their execution plans that is later carried out by the engine. Simply timing transactions to arrive to the database engine at the same time may not always result in a significantly higher frequency of database deadlocks.

However, further analysis of our results shows that the variance of the measured numbers of reported database deadlocks for the approach with scheduling transactions using the controller is much smaller when compared with the variance using the artificial injection approach. We think that the main reason for it is that scheduling transactions to arrive to the database engine at the same time increases the probability that low-level relational algebra operators that form a hold-and-wait cycle may execute at the same time.

When we study the values for the time it takes to reproduce the first occurrence of a database deadlock, the approach with scheduling transactions using the controller takes less MTTDD when compared with the MTTDD using the artificial injection approach for the experiment with a larger number of clients. Increasing the number of concurrent operations that have hold-and-wait cycles and timing them to arrive to the database engine at the same time makes it much quicker to reproduce the first occurrence of a database deadlock. This conclusion may be useful for stress and load testing of DCAs, since it specifies a direction with which it is likelier to cause database deadlocks, thus finding this abnormal behavior quicker and with fewer resources.

## VI. Related Work

Language-based approaches offer different type systems and annotation facilities for programmers to annotate programs, so that type checkers can analyze and detect deadlocks [4], [12]. Given that DCAs contain embedded SQL statements, this approach requires a combination of type systems: one of SQL and the other of the host language in which DCA is written. We are not aware of any language-based approach that can be currently applied to reproduce database deadlocks.

Some approaches use static program analysis to obtain information about deadlocks. RacerX is a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks [9]. Williams et al. [34] defined a deadlock detection algorithm for Java libraries. In contrast with our method, these approaches derive lock graphs directly from Java and C++ source code. These approaches are not currently applicable to reproduce database deadlocks.

Dynamic approaches use runtime data to infer where deadlocks may occur or determine how to predict and resolve them in future program runs. An approach called Dimmunix "immunizes" programs against deadlocks by collecting deadlock patterns, which are subsets of control flow traces that lead to deadlocks [19]. It uses detected hold-and-wait cycles to prevent database deadlocks, but unlike STEPDAD Dimmunix is not designed to reproduce them.

Rx is a dynamic approach that rolls back an application once a deadlock occurs to a checkpoint and retries it again with the hope that the deadlock will be avoided in subsequent executions [26]. A recent work on MagicFuzzer describes a dynamic deadlock detection technique for C++ programs, where MagicFuzzer uses runtime information to prune the number of choices that may lead to deadlocks [5]. A dynamic approach called Sammati provides automatic deadlock detection and recovery for POSIX threaded applications [25]. Unlike Rx, MagicFuzzer, and Sanmati, STEPDAD reproduces database deadlocks instead of fixing them.

Pike is a concurrency bug detector that automatically identifies when an execution of a program triggers a concurrency bug [10]. Pike is related to STEPDAD in that it uses a scheduler to control thread interleaving by intercepting certain library calls and forcing a thread to run at a time that is randomly chosen by the scheduler in an attempt to reproduce a bug. In that, Pike is complementary to STEPDAD, which can use ideas from Pike to improve its scheduler. Unlike Pike, STEPDAD deals with database deadlocks, and it is unclear how Pike can be extended to handle such deadlocks.

Approaches for preventing deadlocks using transactional memory are gaining increasing popularity [20], [33]. Unfortunately, database deadlocks often occur in the distributed setting where there is no shared memory. In contrast to STEPDAD, these approaches may be applicable to reproduce deadlocks among different threads within the same process, but not for distributed environment where external databases and DCAs could be located on different computers.

## VII. Conclusion

We created a novel approach for *Systematic TEsting in Presence of DAtabase Deadlocks (STEPDAD)* that enables testers to instantiate database deadlocks in applications with a high level of automation and frequency. We implemented STEPDAD and experimented with three applications. STEPDAD reproduced a number of database deadlocks in these applications that is bigger by more than an order of magnitude on average when compared with the number of reproduced database deadlocks using the baseline approach. In some cases, STEPDAD reproduced a database deadlock after running an application only two times, while no database deadlocks were reproduced after ten runs using the baseline approach.

## Acknowledgments

We are grateful to our anonymous reviewers and to the PC chairs who provided valuable feedback that helped us improve the quality of this paper. We warmly thank Tathagata Dasgupta, a graduate student from the UIC who contributed to STEPDAD as part of his work towards the completion of the Master of Science degree. We also thank graduate students from the University of Illinois at Chicago who developed subject applications as part of taking a graduate course. We are grateful to the anonymous managers from Fortune 100 companies for their relevant and useful comments and suggestions about testing for database deadlocks. This work is supported by NSF CCF-0916139, CCF-1017633, and CCF-1217928.

## References

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, Nov. 1987.

[2] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 37–40, New York, NY, USA, 2006. ACM.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[5] Y. Cai and W. K. Chan. Magicfuzzer: scalable deadlock detection for large-scale applications. ICSE 2012, pages 606–616, Piscataway, NJ, USA, 2012. IEEE Press.

[6] R. Cochrane, H. Pirahesh, and N. M. Mattos. Integrating triggers and declarative constraints in sql database sytems. VLDB '96, pages 567–578, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[7] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[8] K. R. Dittrich, A. M. Kotz, and J. A. Mülle. An event/trigger mechanism to enforce complex consistency constraints in design databases. *SIGMOD Rec.*, 15(3):22–36, Sept. 1986.

[9] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[10] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 215–228, New York, NY, USA, 2011. ACM.

[11] H. Garcia-Molina. A concurrency control mechanism for distributed databases which use centralized locking controllers. In *Proceedings of the Fourth Berkeley Workshop on Distributed Databases and Computer Networks*, pages 113–122, Berkeley, CA, USA, Aug. 1979.

[12] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. 7th ACM SIGPLAN TLDI '11, pages 15–28, New York, NY, USA, 2011. ACM.

[13] S. Golubchik and A. Hutchings. *MySQL 5.1 Plugin Development*. Packt Publishing, Aug. 2010.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[15] J. Griggs. Database deadlock avoidance patterns. *http://c2.com/cgi/wiki?DatabaseDeadlockAvoidancePatterns*, Sept. 2003.

[16] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Found. Trends databases*, 1(2):141–259, Feb. 2007.

[17] M. Hofri. On timeout for global deadlock detection in decentralized database systems. *Inf. Process. Lett.*, 51:295–302, September 1994.

[18] S. S. Isloor and T. A. Marsland. The deadlock problem: An overview. *Computer*, 13(9):58–78, Sept. 1980.

[19] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. OSDI'08, pages 295–308, Berkeley, CA, USA, 2008. USENIX Association.

[20] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. SPAA '08, pages 297–303, New York, NY, USA, 2008. ACM.

[21] D. Lee, W. Mao, H. Chiu, and W. W. Chu. Designing triggers with trigger-by-example. *Knowl. Inf. Syst.*, 7(1):110–134, Jan. 2005.

[22] D. B. Lomet. Subsystems of processes with deadlock avoidance. *IEEE Trans. Software Eng.*, 6(3):297–304, 1980.

[23] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, Dec. 1986.

[24] M. Nonemacher. Deadlocks in j2ee. *Java Dev. Journal*, Apr. 2006.

[25] H. K. Pyla and S. Varadarajan. Avoiding deadlock avoidance. PACT '10, pages 75–86, New York, NY, USA, 2010. ACM.

[26] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergiesa safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3), Aug. 2007.

[27] S. K. Rahimi and F. S. Haug. *Distributed Database Management Systems: A Practical Approach*. Wiley-IEEE Computer Society Pr, New York, NY, USA, 1st edition, Aug. 2010.

[28] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[29] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[30] A. P. Siebes, M. H. Voort, and M. L. Kersten. Towards a design theory for database triggers. Technical report, Amsterdam, The Netherlands, The Netherlands, 1992.

[31] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[32] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *ICST*, pages 200–209, 2011.

[33] H. Volos, A. J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. ASPLOS '12, pages 211–222, New York, NY, USA, 2012. ACM.

[34] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.