Efficient Regression Tests for Database Applications

Florian Haftmann[1]   Donald Kossmann[1,2]   Alexander Kreutz[1]

[1]i-TV-T AG
Betastraße 9a
D-85774 Unterföhring
{*lastname*}*@i-TV-T.de*

[2]ETH Zürich
Dept. of Computer Science
CH-8092 Zürich
kossmann@inf.ethz.ch

Abstract

If you browse through the articles of www.junit.org, you will find only one article that contains the word *database* in its abstract. This observation is shocking because, of course, testing is just as important for database applications as for any other application. The sad truth is that JUnit simply does not work for database applications, and there are no alternatives on the market place. The reason is that there are some fundamental issues in automatizing regression tests for database applications. This paper addresses one particular issue that arises from the fact that you change the state of a database application while you test it.

When you observe a system, you change the system.
Werner Heisenberg (1901-1976)

1   Introduction

Database applications are becoming increasingly complex. They are composed of many components and stacked in several layers. Furthermore, most database applications are subject to constant change; for instance, business processes are re-engineered, authorization rules are changed, components are replaced by other more powerful components, or optimizations are added in order to achieve better performance for a growing number of users and data. The more complex an application becomes, the more frequently the application and its configuration must be changed.

Unfortunately, changing a database application is very costly. The most expensive part is to carry out

tests in order to ensure the integrity of the application after the change has been applied. In order to carry out tests, most organizations have test installations of all their software components and special test database instances. Furthermore, companies make use of a variety of tools that support regression testing; the most popular tool is the JUnit framework that was developed to carry out regression tests for Java applications [1, 4]. The advantages of regression testing have also been quantified in several empirical studies [10, 6]. Unfortunately, however, testing a database application cannot be carried out automatically using these tools and, thus, requires a great deal of manual work.

The reason for the need of manual work is that the current generation of regression test tools has not been designed for database applications. All the tools we are aware of have been designed for *stateless* applications. In other words, these tools assume that test runs can be executed in any order. For database applications this important assumption does not hold: a test run might change the test database and, thus, impact the result of another test run. The only specific work on testing database applications is [3]. That work gives a framework for testing DB applications. Furthermore, RAGS [7] has been devised in order to test database systems (e.g., SQL Server, Oracle, DBS), but not DB applications.

This work was motivated by a project carried out for Unilever, one of the big players in the consumer goods industry (foods, personal care, home care). Unilever uses a sophisticated e-Procurement application (called BTell by i-TV-T AG) that is connected to a variety of different other applications, including an ERP system (i.e., SAP R/3). Furthermore, Unilever has a portal and provides Web-based access to the e-Procurement application for its own employees and its suppliers. The whole IT infrastructure is currently under dramatic change: software components of different business units are harmonized, new modules of different vendors are added and newly customized, and new business processes involving external users such as suppliers and customers are implemented. These changes are carried out gradually in an evolutionary
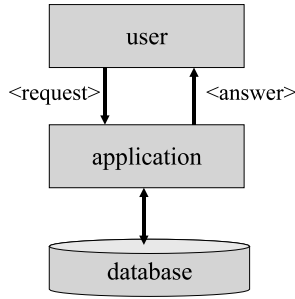
Figure 1: Architecture of Database Applications

way over the next years. In order to carry out regression tests, this company uses a commercial tool, called HTTrace. This tool has a teach-in component in order to record test runs (a sequence of user actions on the application) and a batch component in order to automatically run test runs and detect changes in the answers produced by the application for each user action. For Unilever, we designed the regression test methodology and extended the test tool by a control component that controls in which order the test runs are executed and when the test database is reset.

The remainder of this paper is organized as follows: Section 2 describes the process of database application regression tests in more detail. Section 3 contains basic control algorithms. Section 4 devises more sophisticated algorithms. Sections 5 and 6 present the results of performance experiments. Section 7 concludes this work and proposes avenues for future work.

2   DB Application Regression Tests

2.1   Overview

Figure 1 shows how users interact with a database application. The application provides some kind of interface through which the user issues requests, usually a GUI. The application interprets a request, thereby issuing possibly several requests to the database. Some of these requests might be updates so that the state of the database changes; e.g., a purchase order is entered or a user profile is updated. In any event, the user receives an answer from the application; e.g., query results, acknowledgments, and error messages.

The purpose of regression tests is to detect changes in the behavior of an application after the application or its configuration has been changed. Here, we focus on so-called black-box tests; i.e., there is no knowledge of the implementation of the application available [8]. As shown in Figure 2, there are two phases. In the first phase (Figure 2a), test engineers or a test case generation tool create test cases. In other words, *interesting* requests are generated and issued to a regression test tool. The regression test tool forwards these requests to the application and receives an answer from the application for each request, just as in Figure 1. During Phase 1, the application is expected

to work correctly so that the answers returned by the application are correct and the new state of the test database is expected to be correct, too. The regression test tool stores the requests and the correct answers. For complex applications, many thousands of such requests (and answers) are stored in the repository. If desired, the regression test tool can also record the new state of the test database, the response times of the requests, and other quality parameters in the repository. The regression test tool handles error messages that are returned by the application just like any other answer; this way, regression tests can be used in order to check that the right error messages are returned.

After the application has changed (e.g., customization or a software upgrade), the regression test tool is started in order to find out how the changes have affected the behavior of the application. The regression test tool re-issues automatically the requests recorded in its repository to the application and compares the answers of the updated application with the answers stored in the repository. Possibly, the tool also looks for differences in response time and for inconsistencies in the test database. At the end, the regression test tool provides a report with all requests that *failed*; failed requests are requests for which differences were found. An engineer uses this report in order to find bugs and misconfigurations in the application. If the differences are intended (no bugs), then the engineer uses this report in order to update the repository of the regression test tool and record the new (correct) behavior of the application.

Usually, several requests are bundled into *test runs* and failures are reported in the granularity of test runs. For instance, a test run could contain a set of requests with different parameter settings that test a specific function of the application. Bundling requests into test runs improves the manageability of regression tests. If a function is flawed after a software upgrade, then the corresponding test run is reported, rather than reporting each individual failed request. Furthermore, bundling series of requests into test runs is important if a whole business process, a specific sequence of requests, is tested.

For database applications, the test database plays an important role. The answers to requests strongly depend on the particular test database instance. Typically, companies use a version of their operational database as a test database so that their test runs are as realistic as possible. As a result, test databases can become very large. Logically, the test database must be reset after each test run is recorded (Phase 1) and executed (Phase 2). This way, it is guaranteed that all failures during Phase 2 are due to updates at the application layer (possibly, bugs). Sometimes, companies use several test database instances in order to test different scenarios. Without loss of generality, we assume in this paper that only one test database

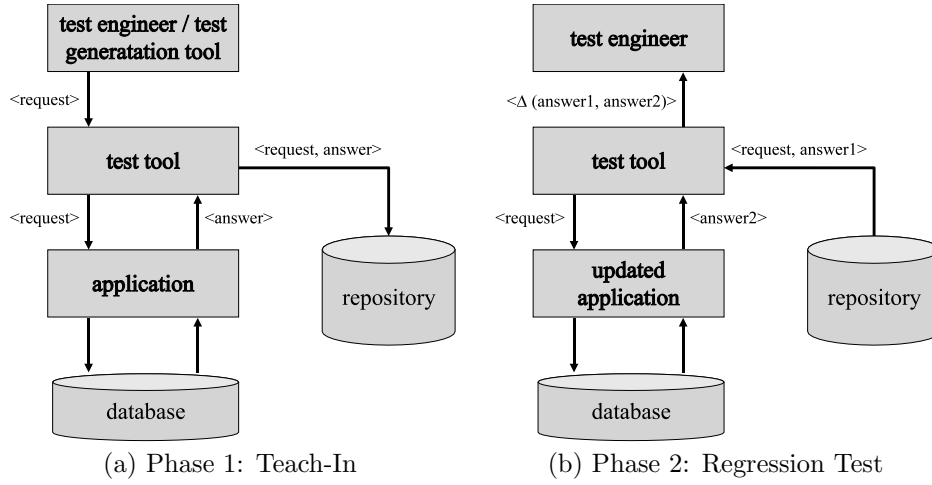(a) Phase 1: Teach-In   (b) Phase 2: Regression Test

Figure 2: Regression Tests

instance is used.

In theory, a test run is okay (does not fail), if all its requests produce correct answers and the state of the test database is correct after the execution of the test run. In this work, we relax this criterion and only test for correctness of answers. The reason is that checking the state of the test database after each test run can be prohibitively expensive and is difficult to implement for black box regression tests. Furthermore, in our experience with real applications, this relaxed criterion is sufficient in order to carry out meaningful regression tests. For checking the integrity of the test database, the interested reader is referred to previous work on change detection of databases, e.g., [2]. If necessary, the techniques presented in that work can be applied in addition to the techniques proposed in this work.

2.2 Definitions and Problem Statement

Based on the observations described in the previous subsection, we use the following terminology:

Test Database $\mathcal{D}$**:** The state of an application at the beginning of each test. In general, this state can involve several database instances, network connections, message queues, etc. For the purpose of this work, we assume that the whole state is captured in a single database instance.

Reset $\mathcal{R}$**:** An operation that brings the application back into state $\mathcal{D}$. Since testing changes the state of an application, this operation needs to be carried out in order to be able to repeat tests. Depending on the database systems used (and possibly other stateful components of the application), there are several alternative ways to implement $\mathcal{R}$; in any event, $\mathcal{R}$ is an expensive operation. In the experimental setup of Section 5, the reset operation took two minutes. Resetting a database involves reverting to a saved copy of the database and restarting the database server process, thereby flushing the database buffer pool.

Request Q**:** The execution of a function of the application; e.g., processing a new purchase order or carrying out a report. For Web-based applications, a request corresponds to a *click* in the Internet browser. Formally, we model a request Q as a pair of functions:

- a_Q: database $\rightarrow$ answer. The a_Q function computes the result of the request as seen by the user (e.g., a Web page), depending on the state of the application.

- d_Q: database $\rightarrow$ database. The d_Q function computes the new state of the application.

Note that Q encapsulates the parameter value settings of the request.

Test Run T**:** A sequence of requests $Q_1, \ldots, Q_n$. A test run, for instance, tests a specific business process which is composed of several requests. Just like requests, test runs can be modelled by a pair of functions, a_T and d_T. a_T computes the set of answers returned by each request of T. d_T is defined as the composition of the d_{Q_i} functions.

Schedule S**:** A sequence of test runs and reset operations. Typically, regression testing involves executing many test runs and reset operations are necessary in order to put the application into state $\mathcal{D}$ before a test run is executed.

Failed Test Run: An execution of a test run in which at least one of the requests of the test run returns a different answer than expected. Failed test runs typically indicate bugs in the application. (As mentioned in the previous subsection, the state of the database at the end of a test run is not inspected.)

False Negative: A test run fails although the behavior of the application has not changed (and there is no bug). One possible cause for a false negative is that the application was in the wrong state at the beginning of the execution of the test run. False negatives are very expensive because they trigger engineers looking for bugs although the application works correctly.

False Positive: A test run does not fail although the application has a bug. Again, a possible explanation for false positives is that the application is in a wrong state at the beginning.

Problem Statement:
The problem addressed in this paper is the following. Given a set of test runs find a schedule such that:

- The schedule can be executed as fast as possible; i.e., the number of $\mathcal{R}$ operations in the schedule is minimized.

- There are no (or very few) false negatives.

- There are no (or very few) false positives.

Unfortunately, as will be shown, there is no perfect solution: an approach that minimizes the number of resets might be the cause for false positives. The purpose of the paper is to find practical approaches that meet all three goals as well as possible.

3 Basic Control Strategies

Going back to the problem statement of Section 2.2, the purpose of this work is to find good *control strategies*. A control strategy determines the schedule and, thus, carries out the following decisions:

1. In which order are the test runs executed?

2. When is the database reset ($\mathcal{R}$ operation)?

This section presents basic control strategies. In principle, these strategies can be classified by the way they deal with false negatives. There are two possible alternatives: (a) avoidance and (b) resolution. *No-Update* and *Reset Always* are representatives for avoidance-based strategies. *Optimistic* and *Optimistic++* are representatives for resolution-based strategies.

3.1 No-Update

The first approach is to create test runs in such a way that they leave the test database unchanged. In other words, the d_T function of each test run T must be the identity function. For example, if a test run has a request in order to insert a purchase order, that test run must also contain a request in order to delete the new purchase order. This policy results in the following *do-nothing* control strategy:

1. Execute the test runs in any order.

2. Never reset the test database.

This strategy was used by Unilever initially because it allows the use of existing regression test tools without any change. On the negative side, however, this approach requires that test engineers know the application very well and exercise a great deal of discipline when they create test runs. A test run that breaks the convention causes false negatives on other test runs and, these, are very expensive to resolve. Furthermore, not all operations of an application can be *undone*; in BTell and SAP R/3, for instance, a completed business process cannot be rolled back. As a result, functions that cannot be undone need to be tested separately, thereby significantly increasing the amount of manual work and cost of carrying out regression tests. The *No-Update* approach is never applicable for applications that implicitly record all user interactions for personalization; an example for such an application is *amazon.com* or any other Web portal. As a result, this strategy was abandoned and will not be studied any further in this work.

3.2 Reset Always

The simplest control strategy that does not require special attention from test engineers and is generally applicable operates as follows:

1. Execute the test runs in any order.

2. Reset the test database before the execution of each test run.

In other words, this strategy carries out regression tests in the following way:

$$\mathcal{R}\ T_1\ \mathcal{R}\ T_2\ \mathcal{R}\ T_3 \ldots \mathcal{R}\ T_n$$

3.3 Optimistic

Obviously, the *Reset Always* strategy is sub-optimal because it involves n resets, if n is the number of test runs. The key observation that motivates the *Optimistic* control strategy is that many of these resets are unnecessary. Resets are not necessary after the execution of a test run that does not involve any updates. Furthermore, a reset between test run T_i and T_{i+1} is not necessary in the following example: T_i tests Module A (e.g., human resources) and possibly updates data used by Module A. Test run T_{i+1} tests Module B (e.g., order processing) such that the updates of T_i are immaterial for the execution of T_{i+1}. These observations give raise to the following *Optimistic* control strategy:

1. Execute the test runs in any order.

2. Whenever a test run fails (i.e., returns different answers), reset the test database, and re-run that test run. Only if the test run fails again, report this test run as a failure.

As an example, the *Optimistic* control strategy could result in the following schedule.

$$\mathcal{R}\ T_1\ T_2\ T_3\ \mathcal{R}\ T_3\ T_4 \ldots T_n$$

In this example, the first execution of T_3 failed because T_1 or T_2 or a combination of both changed data in the test database that was needed for T_3.

In theory, it is possible that the *Optimistic* control strategy results in *false positives*. This situation arises, if, say, T_2 changes the test database in such a way that a bug in the application that is relevant for T_3 is *skirted*. In practice, this phenomenon never happens: for the applications we studied, we were not even able to manually construct a case with a false positive. To be on the safe side, however, we recommend using random permutations of the test runs and changing these random permutations periodically (e.g., every month).

3.4 Optimistic++

The trade-offs between the *Reset Always* and *Optimistic* strategies are straightforward. The *Reset Always* strategy carries out unnecessary resets; on the other hand, the *Optimistic* strategy carries out some test runs twice (e.g, T_3 in the example of the previous subsection). The idea of the *Optimistic++* strategy is to remember invalidations and, therefore, to avoid the execution of test runs twice. In other words, the *Optimistic++* strategy behaves exactly like the *Optimistic* strategy for the first time a regression test is executed (or whenever the permutation is changed), but avoids double execution of test runs in later iterations. Continuing the example from the previous subsection, the *Optimistic++* policy produces the following schedule for the second and later iterations:

$$\mathcal{R}\ T_1\ T_2\ \mathcal{R}\ T_3\ T_4 \ldots T_n$$

3.5 Discussion

The *Optimistic++* strategy shows that simple ideas can improve the performance of database regression tests significantly. The *Optimistic++* strategy shows better performance in all cases than both the *Reset Always* and *Optimistic* strategies. Nevertheless, as will be shown in the next section, it is possible to achieve even better performance than the *Optimistic++* strategy by re-ordering the sequence in which the test runs are executed. Since resetting the test database is a very expensive operation (order of minutes), performance is indeed a critical aspect. A poor control strategy limits the number of test runs that can be applied in regression tests and it limits the size of the test database, thereby resulting in insufficient testing and poor quality database applications (bugs, long response times, etc.).

4 Progressive Algorithms

The previous section described several basic control strategies. This section contains the description of several more sophisticated heuristics in order to decide in which order to execute test runs and when to reset the test database. All these heuristics extend the *Optimistic++* approach; i.e., they only reset the database if necessary and avoid double execution of test runs as much as possible. The basic idea is to *progressively* change the order in which test runs are executed based on the experience obtained in previous regression tests. For example, if one day during the nightly regression tests, test run T_2 was executed *after* test run T_1 and the test database had to be reset for T_2, then the next day, T_2 will be executed *before* T_1. In general, the progressive algorithms *learn* which test runs are in *conflict*. Based on this conflict information, these algorithms determine an order of test runs with as few resets as possible. Typically, the conflict information is not available initially and must be learned by observing the execution of regression tests. If test engineers are willing and able to provide this information or some of this information, then the progressive algorithms converge faster and find a good order of the test runs sooner.

4.1 Slice

4.1.1 Basic Idea

The *Slice* approach re-orders whole sequences of test runs that can be executed without a reset; these sequences are called *slices*. For illustration purposes, we use an example with five test runs: $T_1, \ldots, T_5$. In this example, test run T_1 overwrites data in the test database that is read by T_3; in other words, the test database must be reset if T_1 is executed before T_3. We say that T_1 and T_3 are in conflict and write $T_1 \rightarrow T_3$. Furthermore, T_3 overwrites data that is read by T_2 and T_5; i.e., $T_3 \rightarrow T_2$ and $T_3 \rightarrow T_5$. Initially, the regression test tool does not have this information.

At the beginning, *Slice* behaves exactly like *Optimistic* and executes the test runs in a random order. In the example, *Slice* produces the following schedule with two resets because of $T_1 \rightarrow T_3$ and $T_3 \rightarrow T_5$:

$$\mathcal{R}\ T_1\ T_2\ T_3\ \mathcal{R}\ T_3\ T_4\ T_5\ \mathcal{R}\ T_5$$

Slices are all sequences of test runs that do not fail. In this example, $\langle T_1 T_2 \rangle$ is the first slice, $\langle T_3 T_4 \rangle$ is the second slice, and $\langle T_5 \rangle$ is the third slice. At this point, the algorithm does not know exactly why the first execution of T_3 after $\langle T_1 T_2 \rangle$ fails; the reason could be that T_1 or T_2 overwrite data used in T_3 or that the combination of executing T_1 and T_2 results into a state of the test database that causes a failure of T_3. In any

event, based on the knowledge *Slice* has, there is hope that if T_3 is executed before both T_1 and T_2, then no reset is necessary. Likewise, there is hope that if T_5 is executed before T_3 and T_4, then no further reset will be needed. Furthermore, *Slice* executes T_4 directly after T_3 and T_2 directly after T_1, knowing that this went well in the previous iteration. In other words, *Slice* re-orders whole slices, rather than individual test runs. As a result, *Slice* next tries to execute the test runs in the following order: $T_5T_3T_4T_1T_2$. This is the only ordering of the slices for which there is hope that no reset is necessary. Again, at this point, the algorithm does not know yet that there is a conflict between T_3 and T_2. Using an *Optimistic++* approach, this order to execute the test runs results in the following schedule due to $T_3 \rightarrow T_2$:

$$\mathcal{R} \; T_5 \; T_3 \; T_4 \; T_1 \; T_2 \; \mathcal{R} \; T_2$$

As a result, the following two slices are identified: $\langle T_5T_3T_4T_1 \rangle$ and $\langle T_2 \rangle$. In the next iteration, the algorithm tries to execute the second slice before the first slice. This attempt results in the following *perfect* schedule without any resets, after the initial reset:

$$\mathcal{R} \; T_2 \; T_5 \; T_3 \; T_4 \; T_1$$

Obviously, it is not always possible to achieve such perfect schedules. This is illustrated in the following example with cyclic conflicts:

$$T_1 \rightarrow T_2, T_2 \rightarrow T_3, T_3 \rightarrow T_1$$

For this example, *Slice* starts again with a random order and produces, e.g., the following schedule:

$$\mathcal{R} \; T_1 \; T_2 \; \mathcal{R} \; T_2 \; T_3 \; \mathcal{R} \; T_3$$

Thus, there are three slices: $\langle T_1 \rangle$, $\langle T_2 \rangle$, and $\langle T_3 \rangle$. Next the following schedule is produced:

$$\mathcal{R} \; T_3 \; T_2 \; T_1 \; \mathcal{R} \; T_1$$

The resulting slices are: $\langle T_3T_2 \rangle$ and $\langle T_1 \rangle$. At this point, *Slice* does not attempt any further re-orderings because it knows from the first attempt that there is a conflict between T_1 and T_2 so that executing slice $\langle T_1 \rangle$ before slice $\langle T_3T_2 \rangle$ does not result in any improvement. Following the *Optimistic++* approach, in the next and all later iterations, the following schedule is produced:

$$\mathcal{R} \; T_3 \; T_2 \; \mathcal{R} \; T_1$$

When the algorithm does not attempt any further re-orderings, we say that the algorithm has converged.

4.1.2 Algorithm

The *Slice* algorithm is shown in Figure 3. The algorithm gets a sequence of slices and a conflict database

```
Input:    sequence of slices ⟨s₁⟩⟨s₂⟩⟨sₙ⟩
          conflict database c :: slice × test run → bool
Output: new sequence of slices

/* iterate over all slices starting at Slice 2 */
int m := 2
while m ≤ n do
        /* can we move this slice to an earlier point */
        if ∃k < m : movable(c, ⟨sₖ⟩, ⟨sₘ⟩) then
            k := max{k < m|movable(c, ⟨sₖ⟩, ⟨sₘ⟩}
            move(m, k)
        fi
        m := m +1
od
```

Figure 3: Slice Algorithm

$$\text{movable}(c, \langle s_1 \rangle, \langle s_2 \rangle) = \neg \exists t \in \langle s_1 \rangle : c(\langle s_2 \rangle, t)$$

Figure 4: Criterion to Move $\langle s_2 \rangle$ Before $\langle s_1 \rangle$

as input. At the beginning, there is only one slice that contains all test runs in a random order. For every other iteration, the sequence of slices is defined by the schedule of the previous iteration. The conflict database is used in order to re-order slices. If it is known that T_1 is in conflict with T_2 (i.e., $T_1 \rightarrow T_2$), then a slice that contains T_1 is not moved directly before a slice that contains T_2. Technically, the conflict database implements a function that gets a slice $\langle s \rangle$ and a test run t as input and returns true, if it is known that a sub-sequence of test runs in $\langle s \rangle$ is in conflict with t. For instance, for the following schedule

$$\mathcal{R} \; T_1 \; T_2 \; T_3 \; \mathcal{R} \; T_3$$

$\langle T_1T_2 \rangle \rightarrow T_3$ is entered into the conflict database and the conflict database returns *true* if it is called for, say, the slice $\langle T_1T_4T_2 \rangle$ and test run T_3, since $\langle T_1T_2 \rangle$ is contained in $\langle T_1T_4T_2 \rangle$. At the beginning, $c(\langle s \rangle, t)$ returns *false* for all slices $\langle s \rangle$ and test runs t because there are no conflicts recorded in the conflict database initially. (Details of the implementation of the conflict database are given in Section 4.3.)

The *Slice* algorithm of Figure 3 works as follows. For each slice $\langle s_m \rangle$, it tries to move it before one of its predecessor slices in order to avoid the reset that must be carried out between slice $\langle s_{m-1} \rangle$ and $\langle s_m \rangle$. The exact criterion that determines whether a slice is moved before another slice is given in Figure 4. As mentioned earlier, slice $\langle s_m \rangle$ can be moved before slice $\langle s_k \rangle$ if it can be expected that no reset will be needed between $\langle s_m \rangle$ and $\langle s_k \rangle$. More precisely, slice $\langle s_m \rangle$ can be moved before $\langle s_k \rangle$ if there is no conflict recorded in the conflict database between a sub-sequence of $\langle s_m \rangle$ and a test run in $\langle s_k \rangle$.

$$T_2 \to T_3$$
$$T_3 \to T_1$$

$$\mathcal{R}\ T_1 T_3 T_2$$

Iteration 1: $\quad \mathcal{R}\ T_1\ T_2\ T_3\ \mathcal{R}\ T_3$
Iteration 2: $\quad \mathcal{R}\ T_3\ T_1\ \mathcal{R}\ T_1\ T_2$
Iteration 3: $\quad \mathcal{R}\ T_3\ \mathcal{R}\ T_1\ T_2$

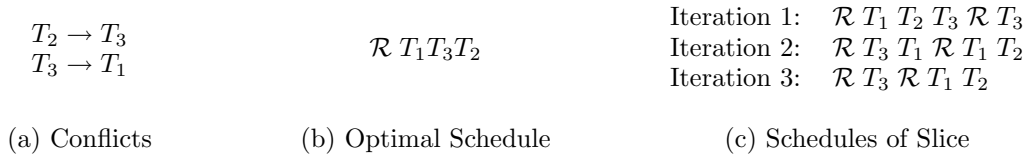(a) Conflicts　　　　　(b) Optimal Schedule　　　　　(c) Schedules of Slice

Figure 5: Sub-optimality of *Slice*

4.1.3　Discussion

The *Slice* heuristics are always better than the basic *Optimistic++* approach. It can be shown that the number of resets decreases monotonically with the number of iterations. In other words, re-ordering slices never results in a worse schedule. Therefore, the *Slice* heuristics always converge after a finite number of iterations. Furthermore, the *Slice* algorithm is very practical. It is simple, requires no extra effort from test engineers, and has very low CPU and memory overhead (Sections 5 and 6). At any time, test runs can be disabled and new test runs can be added and the algorithm continues to operate gracefully. As a safety measure against false positives (Section 3.3), it is possible to restart with a new random permutation of the test runs and discard all the information in the conflict database. Finally, *Slice* is able to consider conflict information that has been provided by test engineers, rather than fully relying on learning this information by observing the execution of test runs.

On the negative side, it is easy to see that *Slice* is not always *optimal*. In the following example, *Slice* produces a schedule with two resets although a perfect schedule with only one reset exists. The conflicts for that example are shown in Figure 5a. The best possible schedule is shown in Figure 5b. The schedules produced by the *Slice* heuristics are shown in Figure 5c.

We tried to extend the *Slice* heuristics by relaxing the convergence condition and by merging slices in order to deal with examples such as the example shown in Figure 5. We do not present the results of these experiments in this paper because our efforts were not fruitful. All the optimal algorithms we could think of had exponential time complexity and were impractical for a number of other reasons (e.g., graceful operation when test runs are added or disabled). Finding a good optimal algorithm and determining whether the problem of finding a schedule that minimizes the number of resets is $\mathcal{NP}$ hard are two avenues for future work.

4.2　Graph-based Heuristics

Since we were not able to find a practical optimal algorithm, we experimented with a set of other heuristics. These heuristics were inspired by the way deadlock resolution is carried out in database systems [9]. The (known) conflicts are represented in a directed graph: the test runs are represented as nodes in the graph, and conflicts are represented by edges. Then, one test run (i.e., node) is selected from the graph according to a certain criterion. This test run is executed first and removed with all its in-going and out-going edges from the graph. The criterion is applied on the resulting graph to find the next test run, and so on.

Details and examples how this approach works can be found in the long version of this paper which is available on the Web pages of ETH Zurich. The long version also describes and evaluates alternative heuristics to select a test run that ought to be executed next. The best heuristics is referred to as *MaxWeightedDiff*. This criterion assigns weights to the edges in the graph, depending on the probability that the two test runs are in conflict. For every test run, *MaxWeightedDiff* computes the sum of the weights of the incoming edges, *fan-in*, and of the outgoing edges, *fan-out*. *MaxWeightedDiff* selects the test run with the largest difference *fan-in – fan-out*. Other criterions only consider the *fan-out* of a test run and/or do not assign weights to edges. These approaches are referred to as *MinFanOut, MaxDiff*, and *MinWeightedFanOut*.

Like *Slice*, these heuristics extend the *Optimistic++* approach and are always at least as good as the pure *Optimistic++* strategy. All the graph-based heuristics converge after a finite number of iterations. Also like *Slice*, none of these heuristics are optimal. In order to compare the trade-offs, we carried out performance experiments which will be presented in Sections 5 and 6.

4.3　Conflict Management

Optimistic++, *Slice*, and all graph-based heuristics require the management of conflict information. As mentioned in Section 4.1, conflicts are recorded in the form $\langle s \rangle \to t$ for Slice and Optimistic++, with $\langle s \rangle$ a sequence of test runs and t a test run. For the graph-based heuristics, edges are represented in the form $s \to t$ which is a special case of $\langle s \rangle \to t$ so that the same management component can be used.

Conflicts are recorded when regression tests are executed and the control strategies learn. Conflict information is needed by the control strategies in order to determine in which order to execute test runs and when to reset the test database. More precisely, the conflict management component must support the following operations:

- **testSub**($\langle s \rangle, t$): Test whether there is a sequence of test runs $\langle s' \rangle$ such that $\langle s' \rangle \to t$ is recorded in the conflict database and $\langle s' \rangle$ is a sub-sequence of $\langle s \rangle$. Sub-sequence is defined as follows: all test runs of $\langle s \rangle$ are also in $\langle s' \rangle$ and they occur in both

sequences in the same order. The *testSub* operation is needed by the *Optimistic++* in order to decide where to place resets and by *Slice* in order to decide if a slice is movable. This operation is also needed when a new conflict $\langle s \rangle \rightarrow t$ is supposed to be inserted into the conflict database. If $testSub(\langle s \rangle, t)$ returns *true*, then the conflict $\langle s \rangle \rightarrow t$ must not be recorded in the conflict database because $\langle s' \rangle \rightarrow t$ which is recorded in the conflict database superimposes $\langle s \rangle \rightarrow t$.

- **findSuper($\langle s \rangle, t$)**: Find all sequence of test runs $\langle s' \rangle$ such that $\langle s' \rangle \rightarrow t$ is recorded in the conflict database and $\langle s \rangle$ is a sub-sequence of s'. This operation is needed in situations when $\langle s \rangle \rightarrow t$ is inserted into the conflict database in order to find all conflicts $\langle s' \rangle \rightarrow t$ which must be removed from the conflict database because they are superimposed by $\langle s \rangle \rightarrow t$.

- **record($\langle s \rangle, t$)**: Insert $\langle s \rangle \rightarrow t$ into the conflict database when it is not superimposed by an existing conflict.

- **delete($\langle s \rangle, t$)**: Remove $\langle s \rangle \rightarrow t$ from the database. This operation is called when it can be deduced that a recorded conflict is not a conflict; for example, if a newly recorded conflict superimposes an existing conflict in the database.

- **weight(t_1, t_2)**: Determine the weight of the edge between test run t_1 and test run t_2 in the conflict graph, as defined in Section 4.2.

- **exists(t_1, t_2)**: Determine whether there is an edge between test run t_1 and test run t_2 in the conflict graph. This function is needed for the *MinFanOut* and *MaxDiff* heuristics.

In order to implement these functions efficiently, all conflicts are organized in a tree. The edges of this tree are labeled with test runs. Paths in the tree, thus, represent sequences of test runs. Nodes of the tree contain sets of test runs. This way, a node of the tree represents a set of conflicts $\langle s \rangle \rightarrow t$, with $\langle s \rangle$ the path of that node and t a test run stored in that node. As an example, Figure 6 shows such a tree for the following conflicts:

$$\langle T_1 \rangle \rightarrow T_3$$
$$\langle T_1 T_2 \rangle \rightarrow T_4$$
$$\langle T_1 T_2 \rangle \rightarrow T_5$$
$$\langle T_1 T_4 \rangle \rightarrow T_5$$
$$\langle T_2 T_3 \rangle \rightarrow T_4$$
$$\langle T_2 T_1 \rangle \rightarrow T_5$$
$$\langle T_2 T_1 T_7 \rangle \rightarrow T_8$$
$$\langle T_2 T_8 \rangle \rightarrow T_6$$

As mentioned at the beginning of Section 3, all approaches behave in the same way when a test run fails
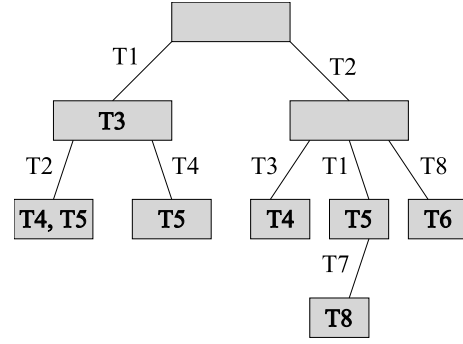

Figure 6: Example: Tree of Conflicts

(i.e., there is a difference in the answer of a request even if the database has been reset). In such an event, the conflict information for that test run is not updated.

Staleness of the information in the conflict database is not a problem. As a result, it is not necessary to adjust the conflict database if test runs are modified, test runs are disabled, new test runs are added, or if the application is upgraded. All these events impact conflicts between test runs. Nevertheless, those events need not be propagated to the conflict database for two reasons: (a) It does not hurt if a conflict is not recorded; for example, all approaches work correctly even if the conflict database is empty. (b) It does not hurt if a *phantom* conflict is recorded in the conflict database that does not exist in reality. Such phantom conflicts might result in extra resets or sub-optimal test run orderings, but all approaches continue to work correctly. In order to deal with the staleness of the conflict database, we recommend scratching it periodically (e.g., once a month). Such an approach is much simpler than trying to keep the conflict database up-to-date all the time.

5 Real-world Experiments

This section contains the results of performance experiments carried out with a real database application (BTell), a commercial regression test tool (HTTrace), and the test database and test runs of Unilever.

5.1 Test Environment

For this set of performance experiments, we used the test environment of Unilever. In particular, we used Unilever's infrastructure to carry out regression tests for its e-Procurement applications (BTell and SAP R/3). Among others, these applications maintain the bill of materials, implement business processes for price and volume negotiation and factory service agreements, and carry out complex forecast reports. Buyers, commercials, and suppliers use these applications through a Web-based interface. The application and the corresponding business processes are still under development and, therefore, regression tests are

102

Approach	Running Time	Resets
Reset Always	189 min	63
Optimistic	76 min	5
Optimistic++	74 min	5
Slice	65 min	2
MaxWeightedDiff	63 min	2

Table 1: Real App: Running Time, Resets

Approach	CPU Overhead	Conflict DB	Iterations
Reset Always	0 sec	0	1
Optimistic	0 sec	0	1
Optimistic++	0 sec	5	2
Slice	< 1 sec	66	3
MaxWeightedDiff	< 1 sec	159	6

Table 2: Real App: CPU, Conflicts, Convergence

crucial. In order to carry out regression tests, Unilever uses a commercial tool, HTTrace. HTTrace works together with Microsoft's Internet Explorer; it records test runs entered by test engineers and is able to automatically apply those test runs and report failures as described in Section 2. A similar capture & replay tool is provided by Microsoft as part of Visual Studio Dot-Net, by IBM as part of Rational Rose, and by Mercury as part of the WinRunner product. Initially, Unilever used a *No-Update* strategy (Section 3.1). The *No-Update* strategy was abandoned, however, because it was impractical from a usability point of view. Therefore, we extended the regression test infrastructure by the other control strategies described in Sections 3 and 4.

Unilever uses dedicated hardware to carry out regression tests. All applications, the database system, and the Web server run on a Sun Enterprise E450 with four 480 MHz CPUs and 1 GB of main memory. Solaris 8 is the operating system of the server. The test client is a PC with a 2.4 GHz Pentium processor running Windows XP. Client and server are connected by a 100 Mbit Ethernet.

For the experiments reported here, the size of the test database was 117.5 MB. There were 63 test runs that were entered by test engineers. These test runs test different aspects of the e-Procurement application. On an average, the test runs contained 80 requests. The shortest test run had 16 requests and the longest test run had 448 requests. All these test runs were entered by test engineers having a No-Update strategy in mind. (Nevertheless, as we will see many test runs violated the basic assumption of that strategy.) Based on the results of this study, Unilever is currently building tools (i.e., robots) in order to automatically generate test runs and achieve a much higher coverage. The goal is to have 10,000 test runs.

In order to implement the reset database operation, several strategies were tested (including an approach that uses the archive logs of the database system). The winning approach was the one that simply remapped the original test database image file into the file system. This approach involved shutting down and restarting the database server and it took about two minutes. Both BTell and SAP R/3 capture all their state in a relational database so that no further actions were required in order to implement the $\mathcal{R}$ operation.

All experiments reported here were carried out with a correct application. In other words, the test tool reported no failures. During the experiments, test runs failed temporarily due to changes in the test database if an *Optimistic* or one of the *progressive* approaches were used. Such failures were handled by resetting the database and re-running the test run, as described in Section 3.3. Re-running a test run never resulted in a failure.

5.2 Results

Table 1 shows how long it took and how many resets were needed in order to carry out the regression tests, depending on the control strategy. The *Reset Always* strategy performed worst; it took more than three hours to carry out the 63 test runs using this strategy because the database was reset before each test run. Clearly, this strategy is not viable if the company wishes to increase the size of its test database and number of test runs by several orders of magnitude because the running time grows linearly with the size of the test database and the number of test runs. All other control strategies performed well in this experiment (running time of about an hour and only few resets). The *Optimistic* strategies worked particularly well in this scenario because test engineers created most test runs having a *No-Update* policy in mind. However, the *Slice* and *MaxWeightedDiff* heuristics performed even better due to re-ordering the test runs so that conflicting test runs were not executed consecutively. We studied the *MaxWeightedDiff* heuristics as the only graph-based heuristics in this experiment because it turned out to be the best graph-based heuristics; we will study the trade-offs of the other graph-based heuristics in Section 6. Virtually, *Slice* and *MaxWeightedDiff* performed equally well in this experiment because they both found schedules with the same number of resets; the difference in running time was due to instabilities in the test environment while the tests were carried out (e.g., network interferance).

Table 2 shows the CPU time to carry out the control strategies (the CPU Overhead column); the overheads are negligible in all cases. Furthermore, Table 2 gives the sizes of the conflict databases for each strategy and the number of iterations it took before the strategy converged. Obviously, the *Reset Always* and *Optimistic* strategies converged after one iteration and recorded no conflicts. The *Optimistic++* strategy always converges after two iterations (Section 3.4) and recorded 5 conflicts in the conflict database in this experiment. *Slice* converged after three itera-

tions and recorded 23 conflicts; *MaxWeightedDiff* converged later and recorded 159 conflicts. In summary, all strategies converged quickly and all strategies except *Reset Always* produced good schedules from the very beginning. Also, the storage requirements and management of the conflict database were negligible in this experiment.

6 Simulation Results

This section presents the results of simulation results that were carried out in order to find out how the alternative control strategies scale. We carried out experiments with more test runs and studied scenarios where more test runs were in conflict. Such scenarios arise, for instance, if test runs are generated automatically or if test engineers do not follow a *No-Update* strategy and do not make sure that all changes are compensated within a test run.

6.1 Test Environment

The simulations were carried out in the following way. First, we generated simple test runs and random conflicts between these test runs by explicitly controlling the data that was read and updated in each test run. Then, we iteratively executed all the test runs using the alternative control strategies, thereby using the expimental set-up described in Section 5.1. We measured up to 100 iterations so that the progressive control strategies were able to learn and converge. In a real-world environment with nightly regression tests, 100 iterations correspond to a period of approximately three months. At the end, we measured the number of resets needed to execute all test runs, the CPU overhead of the control strategies and the size of the conflict database. We repeated each experiment many times (new random permutations to start, new random conflicts) in order to make sure that the 90% confidence intervals are within $\pm 5\%$; the confidence intervals were computed using batch means [5]. In all experiments, the variance was low so that we do not report it here.

In order to study the trade-offs of the alternative approaches, we varied the following parameters:

- **number of test runs:** we carried out experiments with 100 and 1000 test runs.

- **number of conflicts:** we studied scenarios with few and many conflicts. The more conflicts there are, the more difficult it is to find a good ordering of the test runs.

- **distribution:** the distribution controls which test runs are in conflict. In practice, some test runs are update-heavy and are in conflict with many other test runs. An example is a test run that tests the processes that are carried out in a corporation at the beginning of each quarter. For

Approach	10c	100c	1000c	8000c
Reset Always	100	100	100	100
Optimistic/++	3.2	8.6	26.4	83.6
Slice	1.9	3.0	8.5	36.6
MinFanOut	1.1	5.8	21.9	83.4
MaxDiff	1.0	4.3	19.4	79.0
MinWeigthedFanOut	1.4	5.3	24.7	79.5
MaxWeightedDiff	1.0	2.8	17.6	78.8

Table 3: Simulation: Resets
Uniform, 100 test runs, Vary Conflicts

Approach	10c	100c	1000c	10000c
Reset Always	1000	1000	1000	1000
Optimistic/++	3.2	8.7	25.7	80.5
Slice	1.9	2.8	6.7	23.4
MinFanOut	1.5	5.8	19.4	65.8
MaxDiff	1.0	3.4	12.4	52.2
MinWeigthedFanOut	1.3	4.1	15.2	72.3
MaxWeightedDiff	1.0	1.1	3.7	27.9

Table 4: Simulation: Resets
Uniform, 1000 test runs, Vary Conflicts

the simulation experiments, we tested two different distributions:

1. **Uniform:** all test runs are in conflict with other test runs with the same probability.

2. **Zipf:** some test runs are in conflict with many other test runs according to a Zipf distribution.

6.2 Results

6.2.1 Resets

Table 3 shows the average number of resets for each control strategy in a scenario with 100 test runs and a Uniform distribution. We do not show the running time to execute all test runs because the test runs were generated synthetically (consisting of only a few updates and reads) and measuring their running time was not meaningful. The number of conflicts was varied between 10 and 8000. Again, the *Reset Always* strategy performed worst in this experiment: it carried out 100 resets, as many resets as there are test runs. The *Optimistic* strategies performed very well, on an average, if there were few conflicts. The performance of the *Optimistic* strategies degraded radically with an increasing number of conflicts between the test runs. The more conflicts there are, the more important it becomes to re-order the test runs in such a way that conflicting test runs are not executed sequentially. *Optimistic* and *Optimistic++* showed the same performance in this experiment because only the number of resets was counted.

Slice and all graph-based heuristics clearly outperformed *Optimistic* in this experiment. Among the graph-based heuristics, *MaxWeightedDiff* was the

clear winner, outperforming all other graph-based heuristics in all cases. For 10 conflicts, *MaxWeightedDiff* was almost always able to find perfect schedules (1.0 resets); occasionally, there were cycles in the conflicts such that perfect schedules were not possible, but on an average, these cases did not have a noticeable impact. Comparing the performance, of *Slice* and *MaxWeightedDiff*, *MaxWeightedDiff* was clearly better if there were few conflicts (100 or less), whereas *Slice* was better, if there were many conflicts. We observed this behavior in all our experiments. We do not have a comprehensive explanation for this behavior. Analyzing the traces, it seems that *Slice* is good for many conflicts because it acts *positively*; i.e., it keeps non-conflicting test runs together (in slices). On the negative side, *Slice* does a comparably poor job of isolating *bad* test runs which are in conflict with many other test runs. Those test runs should be executed last, but they end up in the middle of slices. On the other hand, the graph-based heuristics act *negatively*: they are able to identify *bad* test runs, but they are not able to keep test runs together and execute them sequentially, if it is known that these test runs are not in conflict.

Table 4 shows the average number of resets for each control strategy in a scenario with a Uniform distribution. Again, the number of conflicts was varied. This time, however, sets of 1000 instead of 100 test runs were studied. The trends are the same as in Table 3: *Reset Always* performed worst, *Optimistic* was next, *MaxWeightedDiff* was the best graph-based approach, and *Slice* was very good if there were many conflicts and worse than *MaxWeightedDiff* if there were few conflicts. Comparing Tables 3 and 4, it can be seen that the number of resets decreased for an increasing number of test runs, if the number of conflicts stays constant. In other words, the *density* of conflicts has a strong impact on the number of resets required. The higher the density, the more difficult it is to find good schedules.

Table 5 shows the average number of resets in a scenario with 1000 test runs, varying the number of conflicts. This time, a Zipf distribution was used. Again, the major observations were the same as in the previous two simulation experiments and the overall results are very similar. Comparing Tables 4 and 5, it can be observed that the results with the Zipf distribution were slightly better than with the Uniform distribution. The reason is that the Zipf distribution concentrates more conflicts to few *bad* test runs. Having those *bad* test runs out of the way, then makes it easier to order the remaining test runs which have fewer conflicts between them.

6.2.2 Convergence Speed

Figure 7 shows how quickly the individual strategies converge. As mentioned in the description of the test

Approach	*10c*	*100c*	*1000c*	*10000c*
Reset Always	1000	1000	1000	1000
Optimistic/++	3.2	8.1	23.1	67.7
Slice	1.9	2.8	6.3	17.8
MinFanOut	1.0	4.7	14.4	61.0
MaxDiff	1.0	3.4	11.2	45.3
MinWeigthedFanOut	1.3	3.9	13.2	54.3
MaxWeightedDiff	1.0	1.1	4.1	31.0

Table 5: Simulation: Resets
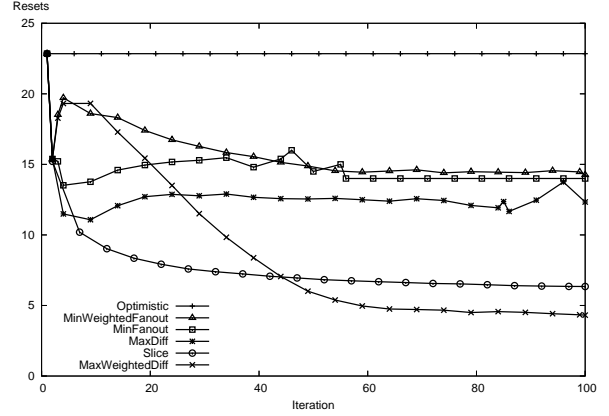Zipf, 1000 test runs, Vary Conflicts

Figure 7: Simulation: Convergence
Zipf, 1000 test runs, 1000 conflicts

environment, we carried out 100 iterations for each set of test runs and then repeated the experiments many times with different sets of test runs in order to compute confidence intervals. In the previous subsection, we reported the average number of resets needed for each control strategy for the 100th iteration. Figure 7 shows the number of resets by iteration, 1 to 100. In this experiment we used 1000 test runs with 1000 conflicts and a Zipf distribution.

In the first iteration, all control strategies behaved in the same way. (An exception was the *Reset Always* strategy which is not shown in Figure 7 because it alway had 1000 resets.) The *Optimistic* approaches did not improve with the number of iterations because these approaches do not learn and do not re-order test runs. (Again, *Optimistic* and *Optimistic++* showed the same performance because we counted the number of resets only.) The other strategies improved with the number of iterations, thereby learning more and more conflicts and adjusting the order correspondingly. In all our experiments, *Slice* improved and converged the fastest. With *Slice*, it took only a few iterations before very good schedules were found. The graph-based heuristics were much slower in the experiment shown in Figure 7 and in all other experiments (not shown here). Furthermore, the graph-based heuristics are bumpy; they sometimes produced worse schedules from one iteration to the next. In this particular exper-

Approach	CPU Time	Conflict DB
Slice	8.0secs	58375.4 nodes
MinFanOut	1.9secs	23004.3 nodes
MaxDiff	3.0secs	46857.6 nodes
MinWeigthedFanOut	3.8secs	56672.4 nodes
MaxWeightedDiff	2.5secs	87466.7 nodes

Table 6: Simulation: Overheads
Uniform, 1000 test runs, 10,000 Conflicts

iment, *MaxWeightedDiff* was better than *Slice* at the end, but it took almost 50 iterations before it did so. In a company with nightly regression tests, this means that *Slice* is better in the first two months. *Slice* is always better if conflict information is scratched on a monthly basis.

6.2.3 Overheads

Table 6 shows the CPU time and the size of the conflict database for each control strategy. These times were recorded for the 100th iteration of an experiment with a Uniform distribution, 1000 test runs, and 10.000 conflicts. These were the highest overheads we could observe in all our experiments. (For the Zipf distribution, the overheads were lower by about a factor of 2; for less conflicts, the overheads were much lower.) Table 6 shows that the running times of the control strategies were only a few seconds, compared to several hours that it takes to actually execute the test runs and to reset the test database for a real commercial application (Section 5). In other words, it is worth-while to use progressive heuristics in order to optimize regression tests because the overhead of the progressive approaches is much lower than the savings these approaches achieve. Furthermore, the size of the conflict database was at most in the order of hundreds of kilobytes, and, therefore, fitted easily into main memory. It is possible that the conflict database records more conflicts than actually exist. Nevertheless, the number of conflicts recorded in the conflict database is always constrained by the square of the number of test runs and is much lower than that upper bound in practice.

7 Conclusion

Regression testing is a well-studied technique in software engineering. The most prominent framework is JUnit. Furthermore, there are several commercial tools which have been developed for a variety of applications and architectures; e.g., WinRunner, IEPad, Jedi, LIXTO, and Andes. Unfortunately and somewhat surprisingly, all these tools do not work well for database applications; these tools only work well if applications are stateless or tests can be designed in such a way that they do not alter the state. The AGENDA framework is the only related work that specifically addresses the testing of database applications [3].

This work presented a general black box approach

in order to carry out regression tests for database applications. Furthermore, it was shown that the order in which test runs are applied is important. Naive approaches to carry out regression tests either do not scale well, or put a heavy burden on test engineers, or limit the number of tests that can be carried out automatically. This paper proposed and evaluated alternative approaches. The evaluation was carried out using a real test scenario of an industrial user (Unilever) and simulations. As a result, two heuristics, *Slice* and *MaxWeightedDiff*, were identified that perform very well, have low overhead, and require no extra effort from test engineers. Between these two heuristics no clear winner could be identified: *MaxWeightedDiff* performs slightly better if there are only few conflicts; on the other hand, *Slice* finds good schedules faster which is important if new test runs are added and the conflict information is scratched periodically. Unilever decided to use the *Slice* heuristics.

The whole topic of testing database applications is still in its infancy. No rigorous methodologies have been devised yet and there are are several open issues such as the automatic generation and evolution of test runs, the generation of test databases, and the development of platform independent tools. We plan to study these challenges as part of future work.

References

[1] K. Beck and E. Gamma. Test infected: Programmers love writing tests. http://members.pingnet.ch/gamma/junit.htm, 1998.

[2] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 26–37, Tucson, AZ, USA, May 1997.

[3] D. Chays, Y. Deng, P. Frankl, S. Dan, F. Vokolos, and E. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verfication and Reliability*, 14(1):1–29, 2004.

[4] JUnit.org. Junit homepage. http://www.junit.org.

[5] A. Law and J. Carson. A sequential procedure for determining the length of a steady-state simulation. *Operations Research*, 27(5):1011–1025, Sept. 1979.

[6] Y. Liu. Regression testing experiments and infrastructure. Oregon State University, Master Thesis, 1999.

[7] D. Slutz. Massive stochastic testing of SQL. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 618–622, New York, NY, USA, Aug. 1998.

[8] I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, MA, USA, 1995.

[9] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 2001.

[10] E. Wong, J. Horgen, S. London, and H. Agrawal. Effective regression testing in practice. In *IEEE Symposium of Software Reliability Engineering (ISSRE 97)*, pages 264–274, Albuquerque, NM, USA, Nov. 1997.