White-Box Testing for Database-driven Applications: A Requirements Analysis

Klaus Haller
COMIT AG,
Pflanzschulstr. 7
CH-8004 Zürich, Switzerland
klaus.haller@comit.ch

ABSTRACT
White-box testing is an important part of every software testing and quality assurance strategy. Testing database-driven applications requires the adoption of white-box testing, but it is not clear what adoption is needed. Instead of focusing on a single problem and a possible solution, this paper elaborates all of the main challenges from a practitioner's view. Starting with a generic testing process, we analyze for each process step whether and, if so, which adoptions are needed, and redefine the concepts of test cases and coverage. We discuss test database state generation methods and the problem of scheduling test cases efficiently. Thereby, we provide a road map for the emerging domain of testing database-driven applications and for making such testing useful for commercial software development.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: *Software/Program Verification,* D.2.5 [Testing and Debugging]: *Testing tools, coverage testing,* H.0 [GENERAL]

Keywords
Information Systems, Databases, Testing, Test Coverage

1. MOTIVATION

After decades of software development practice, companies can still differentiate from competitors by the reliability of their products. Testing is a key issue in reliability and quality assurance. However, the specifics of tests for database-driven applications (DBAPs) are too often neglected because there are no clear guidelines and tools for such tests. To address this challenge and thereby improving our software testing process, we decided to focus first on the DBAP specifics of white-box testing. In general, white-box testing is an attractive option because it allows identifying test cases automatically. Our discussion is based on our experience in two application areas: PL/SQL script development for data migration purposes [1][2] and software development in the area of credit ratings in commercial banking.

We structure our presentation as follows: Section 2 provides an overview of the different testing approaches and tasks in the testing process. Specific challenges of DBAPs are the focus of Section 3. Section 4 focuses on test coverage issues. Section 5 analyzes correctness criteria and test cases for DBAPs. We continue with how to generate test data (Section 6), how to select test cases for regression testing (Section 7), and how to schedule test cases efficiently (Section 8). We conclude our paper with a short discussion and outlook (Section 9).

DBTest'09, June 29, 2009, Providence, Rhode Island, USA.

2. TESTING TASKS AND APPROACHES

There are many reasons for testing DBAPs. For example, load and performance tests check how a DBAP performs if the DBAP is used concurrently by many users. Source code inspection might try to identify potential security leaks. In this paper, we are interested in the functional correctness of a DBAP: whether it returns the specified results. Functional testing approaches are grouped into three types: experience-based, dynamic, and static. *Experience-based testing* relies on the know-how of testers and users. *Static tests* inspect the code without executing it, e.g., to find variables read but not initiated before. *Dynamic testing* invokes the application and checks whether it returns the expected results. The two main dynamic testing approaches are white-box-testing and black-box-testing. *Black-box testing* identifies test cases *without* looking at the source code but by analyzing the specification. *White-box-testing* analyzes the source code, e.g., to identify possible execution paths and parameter sets for the invocation to ensure that the intended execution path is taken. [3]

The testing process for DBAPs (Figure 1) consists of three phases: preparation, execution, and evaluation. The preparation phase comprises the test case identification task and the test data (state) generation task. The execution phase consists of three tasks. The first task is selecting the relevant test cases for regression tests. Executing only "needed" tests lowers costs. Secondly, test cases should be executed in an optimal order to prevent the need for a new database state to be loaded for each test case. Thirdly, it is important to log the execution so that failures can be analyzed more easily. The evaluation phase is formed by the interpretation of test results and the clean-up of the database. In this paper, we concentrate on the tasks requiring an in-depth discussion – all but the evaluation and logging tasks.

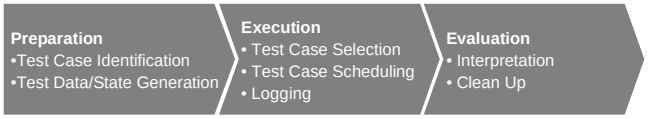

Figure 1: The Testing Process: Phases and Tasks

3. CHALLENGES

With so much existing work on testing, it is important to start with identifying the challenges specific to DBAPs. Therefore, we rely on a generic DBAP model (Figure 2). In the upper section, a procedure symbolizes the application. The procedure has input

and output parameters and uses transient variables. The procedure contains embedded SQL statements that access the database. The SQL engine executes the statements, thereby accessing the database tables. We identified four DBAP-specific challenges.

Challenge 1 (State Challenge): Testing stateless applications means invoking a procedure and checking whether it returns the specified values. DBAPs additionally interact with a database, which stores data such as account balances persistently (Figure 2 ❶). Thus, test cases must also specify the database state before test case execution and the expected state afterwards.

Challenge 2 (Language Layers Challenge): A DBAP can access a database in different ways. Firstly, many databases provide a programming language and execution environment, such as Oracle with PL/SQL. Secondly, the programming language used for application programming might provide a library such as JDBC for database access. Finally, there are persistency frameworks, such as Hibernate. In all cases, there are two layers involved: a "normal" programming language and the database access specification. The coupling and interplay of the two layers requires special attention (❷).

Challenge 3 (Data Parallelism Coverage Challenge): SQL (or XQuery) are set-oriented and inherently carry data parallelism. Executing the SQL statements means accessing, evaluating, and potentially changing many rows in parallel (❸). This requires rethinking coverage criteria for DBAPs.

Challenge 4 (Dynamic Code Challenge): The Java reflection API allows modifying code at run-time. It is rarely used. In contrast, SQL statements are often constructed on-the-fly in DBAPs. This causes difficult problems in general which are outside of the focus of this paper.

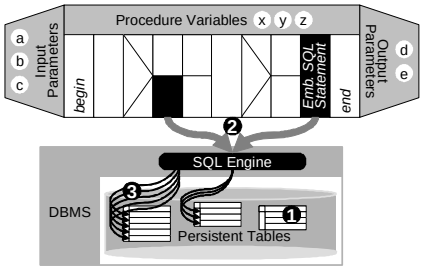

Figure 2: Generic DBAP Model

4. DATA PARALLELISM AND COVERAGE

The quality of test case sets depends not on their size, but on their variety. Variety means that different situations and different parts of the application are tested. In literature, the standard term is "coverage", inspired by the aim of covering all code parts and conditions to be tested [4]. Our discussion is based on the control-flow-based coverage criteria [3]. In short, these criteria represent an application as a directed graph. Test cases are execution paths between the start node and the end node. Nodes are (linear) statement sequences. Edges between nodes are transitions taken if the condition of the edge is fulfilled. Coverage criteria decide whether a set of test cases provides sufficient variety. Three main coverage criteria exist: statement coverage (each node/statement sequence is executed at least once), branch coverage (each edge is used at least once), and path coverage (each possible path from the start node to the end node is taken).

Commercial tools like IBM Rational PurifyPlus [5] allow, for example, checking whether all application code has been executed. To our best knowledge, there is nothing comparable for DBAP coverage criteria, though much effort has been put into identifying suitable criteria. Suggestions are to check whether all tables are accessed, whether all embedded SQL actions have been executed, or whether the flow of data items between different SQL statements is tested [6]. A different approach concentrates on the coverage aspect of single SQL statements and the terms in their WHERE clause respectively their JOIN conditions [7].

Again, we find research concentrating either on complete execution paths or on single statements only. Each concept is too radical to be used alone in practice. The crucial but not obvious point is the data parallelism coverage challenge and the involved effort. We illustrate this in a short example. Let us assume that a bank sends small Christmas presents to customers with bank accounts containing at least 20'000 EUR or to everyone they know who turned 18 or 65 during the year. This translates into the following selection criterion:

```
SELECT p.customerid
FROM   t_persons p
       LEFT OUTER JOIN t_account a
       ON p.customerid=a.customerid
WHERE  YEAR(p.birthday)-YEAR(%TODAY)=18 OR
       YEAR(p.birthday)-YEAR(%TODAY)=65 OR
       a.balance>19999
```

The statement has three terms in the where clause (e.g., `a.balance>19999`) and one term in the join condition (`p.customerid=a.customerid`). There are two obvious coverage criteria:

a) **Elementary query complexity fulfillment:** There is at least one row or row combination such that all where-terms are at least once true and once false.[1] Join conditions must be satisfied at least once and at least once not.

b) **Complete query complexity fulfillment:** There is at least one row combination row for each possible combination of term results. If there are terms t1 and t2, there shall be at least one row for which t1 is true and t2 is true, one row for which t1 is false and t2 is true etc. Join-conditions have five situations to be considered: (I) ANY X/THE SAME X, (II) ANY X/ANY Y, Y≠X, (III) NULL/NULL, (IV) ANY X /NULL, (V) NULL/ANY X.

Figure 3 provides a short example. The upper of the three sections shows two tables with sample data. The middle section consists of a view of a full outer join of the tables involved in the join. The lower section addresses the fulfillment aspect. *Elementary fulfillment* is straightforward. Each term of the WHERE clause referencing attributes of T_ACC respective of T_PERS corresponds to a row in the fulfillment table in the lower part (A, B). If the attribute "Fulfilling rowid" contains a row id, there is at least one row in table T_ACC respectively T_PERS covering this case. Next, all join conditions of the query should be fulfilled at least once by the row combination (C). The *complete fulfillment* table is much more complex. There is one column for each term or join condition (X) and all possible combinations have to be considered (Y).

[1] We informally use the term "row combination". It is a combination of different rows involved in joins. Thus, a row combination row is a row of the full outer join of all involved tables. Certainly the NULL case requires special attention, as a recently emerged discussion shows [8]. However, this question is not a main concern for this paper.

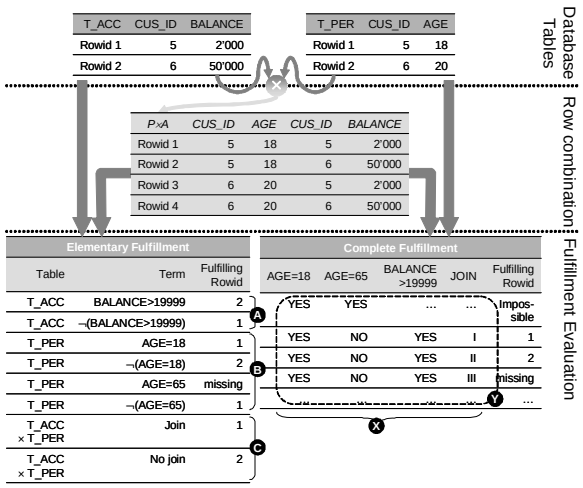

Figure 3: Coverage Fulfillment Example

Elementary fulfillment has linear complexity in terms of needed test cases (the sum of twice the number of selection terms and join conditions), but covers only a few interesting cases. Complete fulfillment looks preferable, but has exponential complexity. The complexity prevents its usage in practical software engineering. DBAPs usually have hundreds of much more complex queries. Even having sample data for each row of the complete fulfillment table is not enough. It is manual work to decide for each row whether it should be in the result set.

The only way we see to address the fundamental dilemma of DBAP testing, the sheer amount of data and manual work needed, is to identify the complex and risky statements (respectively statement sequences). Each of them is tested on its own considering query fulfillment coverage. Additionally, there are normal unit-testing test cases like invoking procedures. They focus on execution path aspects. So, we have two test case types complementing each other. The following section discusses them further by introducing the terminology of "deep" and "shallow" test cases.

5. CORRECTNESS AND TEST CASES

The standard approach for evaluating the functional correctness of stateless functions is based on (stateless) test cases. A stateless test case consists of the procedure to be tested, input parameter values, and the expected output parameter values.

Definition (stateless test case): *A stateless test case TC_{SL} is a triple TC_{SL}=< ρ, Π_I, Π_O> consisting of the procedure ρ to be invoked, input parameter values Π_I, and the expected output values Π_O.*

DBAPs require a modified correctness criterion and an adopted test case concept. Table 1 compiles the different correctness criteria. They are mostly extractions from previously published works of different authors. However, understanding the similarities and differences between them is a major step for understanding why so much different work exists on DBAP testing.

We name the first DBAP correctness criterion **schema correctness**. Schema correctness means that the schema reflects a specification *and* the real world correctly, as intended. In other words, the data model is defined correctly and implemented accordingly. A second criterion is **data conformity correctness.** This criterion assumes that there are two kinds of constraints: those enforced and those not enforced by the database schema.[2] Data conformity correctness checks whether DBAP data violates the latter constraints. Chays et al. [9] identified this criterion and provided a solution.

In our software development projects, a third criterion is the most relevant: **application correctness**. This criterion is based on test cases. We distinguish two types: shallow test cases and deep test cases. Deep test cases focus on the data and data manipulation, e.g., of the correctness of a sequence of SQL statements calculating the interest rates for bank accounts. **Shallow test cases** are on a higher level. They concentrate on the execution of complete execution paths. A sample execution path is the end-of-year processing of a bank. First, the process invokes the calculation of interest payments and ensures that the payments are booked. Then it invokes a service to make out the balance sheet.

More precisely, a shallow test case invokes a procedure ρ with given input parameter values Π_I and constraints Σ_I defining the database state before the invocation. The constraints can be a set of rows (or even all rows) which must be in the database tables, but can also include requirements such as the existence of at least one account for customer 505 or constraints regarding the database and session configuration (e.g., date to char conversion rules or isolation levels). After the execution of the procedure, the output parameter values Π_O are checked against the values stated in the test case specification. In addition, the database state is checked to determine whether it fulfills the constraints Σ_O defined to be valid after the execution of the test case.

This test case and correctness model was introduced by Willmor et al. (see e.g. [10]). It is sufficient for small applications. Complex systems such as core-banking platforms have different needs. It is not sufficient to state that the balance of account 554887 is CHF 1'000 and that the interest rate is 2.5%. Complex systems require seed data. Accounts must have an owner. Interest payments for customers imply corresponding bookings on internal expense accounts etc. A practical useful input database definition has two parts. Firstly, there are hard requirement like "balance of account 554887 is CHF 1'000." Secondly, there is a *suggestion* for a sensible overall database state like "take the database snapshot of January 3rd at 8 p.m." However, other states *might* work as well. Distinguishing between hard entry constraints Σ_I^* (the account balance) and suggested entry constraints Σ_I^+ (system state as of January 3rd) gives additional freedom for scheduling test cases. We take advantage of this distinction in Section 8.

	Criterion	*Focus*
1	**Schema Correctness**	Implemented schema reflects real-world (and data model)
2	**Data Conformity Correctness**	Data reflects not-schema-enforced constraints of the data model
3	**Application Correctness (AC)**	Results as defined in test cases
3a	**- Deep AC**	Table contents after one or more SQL statements
3b	**- Shallow AC**	Return values and side-effects in DB after complete execution path

Table 1: DBAP Correctness Criteria

[2] It is important to be aware that the term *constraint* has two meanings in this paper. Firstly, there are DB constraints such as NOT NULL. Secondly, constraints can be first or second order logical expressions such as, "there is a customer with ID 505".

Definition (shallow test case): *A shallow test case TC_S is a 6-tuple TC_S=< ρ, Π_I, Σ_I^*, Σ_I^+, Π_O, Σ_O> consisting of the procedure ρ to be invoked, input parameter values Π_I, explicit constraints Σ_I^* for the starting database state, implicit constraints Σ_I^+ for the starting database state, the expected output parameter values Π_O, and the expected resulting database state Σ_O .*

Shallow test cases precisely specify an expected end-to-end test case. They are expensive to generate and maintain, especially if there is no tool support. To limit their number (and thereby effort and costs), they should be used only for crucial functionality and not for checking minor variants of calculations.

Deep test cases form the second test case type for DBAPs**.** This idea is based on work on the AGENDA framework [11]. An application is a sequence of one or more set-oriented statements such as UPDATE, DELETE, or SELECT. The statements can access one or more tables. Deep test cases consist of the statements and two constraint sets. The first (qualifying) constraint set Σ_Q defines the data for which the test case is applicable. The qualifying constraints are evaluated for each row before the sequence of statements is executed. Afterwards, all rows that fulfilled the qualifying constraints are checked against the second (evaluation) constraint set Σ_E. Rows fulfilling the qualifying constraints, but not the evaluation constraints, are failures.

Definition (deep test case): *A deep test case TC_D is a triple TC_D=< σ, Σ_Q, Σ_E> consisting of a sequence of statements σ to be performed, the qualifying constraints Σ_Q, and the constraint set Σ_E defining which constraints the qualifying rows must fulfill after the execution of σ.*

To give a short example: A test case shall check the correctness of the interest bookings for all Swiss Francs savings accounts. The interest rate shall be 0.75%. We assume that one table stores all customer accounts and their actual balances. The deep test case is constructed as follows: All booking statements together form σ. The qualifying constraints Σ_Q are (i) that the account type must be savings and (ii) that the currency must be CHF. The evaluation constraint set Σ_E demands that the account balance has been increased by 0.75%.

Deep test cases operate on large data sets. They are helpful for complex statements used for batch-operations such as interest payments during the end-of-year processing. Deep test cases can check different variants of transformations and calculations concurrently. One might imagine using them to check the interest bookings on not only Swiss Francs savings accounts but on all accounts by simply adding additional deep test cases. This is much cheaper than setting up many shallow test cases where suitable accounts must be identified or created and maintained. However, it is crucial for deep test cases to have "varied" data.

6. TEST CASE DATA ENGINEERING

Test case identification corresponds to the construction of constraint sets. Constraint sets ensure the execution of certain execution paths (shallow test cases) or guarantee a certain data parallelism coverage criterion (deep test cases). The next step is to populate the database with fitting data. Some research prototypes exist [10][12], but there are no tools for commercial DBAP development. Thus, we elaborate the consequences for the testing process if the database state is not optimal.

Usefulness of a database state is expressed by four compliance levels: type compliance, schema compliance, application compliance, and path compliance. *Type compliance* demands only that test data to be inserted into tables conforms to the attribute types the columns have, e.g., only numbers are inserted into NUMBER columns. *Schema compliance* requires additionally that all rows respect the schema constraints (NOT NULL, primary-foreign key constraints, etc.). If data is not schema compliant, the database refuses it. Thus, each database state loaded successfully in a database is guaranteed to be schema compliant. *Application compliance* is the third level. Applications do not write and process all schema-compliant data. A core-banking system could, for example, prevent negative balances on saving accounts by implementing conditions in the DBAP code for withdrawals and money transfers, but there are no corresponding schema constraints. Application compliance demands that test data contains only data that could only be inserted and processed by the DBAP. Finally, *path compliance* assures that the data is the right data for executing a certain execution path. If the path requires account 1774552 to have a balance over CHF 100'000, path compliance guarantees it.

Figure 4 illustrates the compliance level hierarchy. Cells of the matrix represent database states (we abstract from input parameters here). All cells together form the universe of possible database states. A cell can represent data that cannot be loaded into the database because it is not schema compliant (–). A cell can also indicate that its execution does not identify a failure (✓), indicate a false positive (↓) or a failure (×). False-positive means that testers *seem* to find a failure, but the failure does not appear in normal application usage. The only reason for this is that the application is used in an unspecified way. For example, if a closed savings account is defined as having a balance of zero, the application code can assume this without checking it. So if the test data contains closed saving accounts with a negative balance, the application might react unexpectedly. Thus, test data must be at least application-compliant to prevent false-positives.

The first goal of testing is to find all failures. A failure in the application code can be detected with different test data constellations. Cells in Figure 4 with "×" allow us to identify one of two failures, one represented by a white background color, the other by a dark grey one. Finding all failures demands the execution of one test case per execution path. If we execute each test case with path compliant data, we find all failures. If we do not have the right data, we do not know (at least not a priori) which execution path we finally execute. Instead of executing path *A*, which assumes five rows in a table, we might take path *B* because the table is empty. If each execution path is not executed at least once, this is a risk for software quality.

Finally, we take a closer look at the approaches for data generation and the compliance level they assure:

- *Tester defined.* A tester manually specifies the test data, e.g., after a code analysis. Thereby, she can ensure path-

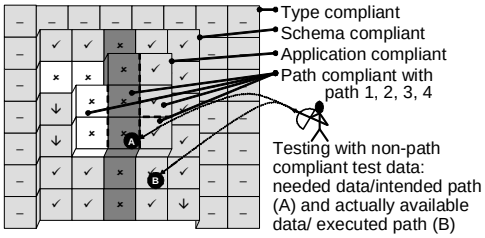

Figure 4: Sample Test Data Universe

compliance. This approach is useful for a few test cases and small data sets.

- *Schema-derived.* Schema definitions restrict the data that can be stored in the table by constraints such as primary-foreign-key-constraints or NOT NULL-constraints. Collecting all constraints from the database catalog allows the generation of schema-compliant data.
- *Path-derived.* Execution paths might require certain data in the database to be defined by constraint sets. A constraint solver solves the constraints and thereby generates path compliant test data automatically. Prototypes with limitations exist [7][10], but we are not aware of any suitable tool for commercial DBAP development projects. Furthermore, Muli-RQP [13] is a convincing approach for generating test database states based on declarative test case specifications.
- *Live-data.* DBAPs in use have live-data in their databases. Live-data is always application-compliant (at least for the current version of the application). Some authors argue strongly against using live-data [14]. We have learned during many projects that using live-data is the cleverest thing one can do, especially for deep test cases. For shallow test cases it might require much manual work to determine that account 10008 and not account 10009 is useful for a certain test case.

7. TEST CASE SELECTION FOR REGRESSION TESTS

When a new application is tested for the first time, it is sensible to execute all test cases. It is sensible to execute all tests before a major release of a DBAP is shipped to customers, too, but only if the test case set is not too large. However, in most cases regression testing is the only choice for cost reasons. Regression testing executes a test case again only if the application has been changed and the change might affect this test case [4]. Reasoning about the potential effect requires (a) analyzing how the application has changed and (b) identifying how changes are linked to test cases. Regression test case selection analyzes the source code to find changes in the case of stateless applications and determines the affected execution paths respective to the corresponding test cases. Willmor and Embury were the first to rethink regression test case selection for DBAPs [15]. In our paper, we want to complement their work with a discussion structured by the different DBAP components and DBAP data:

- *Application code.* If we change the application code, there is a need for retesting the DBAP.
- *Database schema.* If the size of attributes changes (for example, VARCHAR2(100) to VARCHAR2(1000) or to VARCHAR2(10)), the application code is unchanged but the application might crash nevertheless. Adding new attributes can affect e.g. SELECT * FROM T_TAB statements. They return more attributes. If a NOT NULL constraint is added, inputs that succeeded before might fail. In other words, database schema changes usually impact large parts of the application.
- *Application data.* If a bank adds a new customer to the database, there is no need for retests.
- *Metadata.* Systems often store metadata such as the account products a bank offers. If the application code contains fix references, for example, in a workflow (if product_id=500 then …) for savings accounts, this is obviously bad coding practice. But changing the metadata might require regression tests. However, distinguishing between application and metadata is often difficult.

8. TEST CASE SCHEDULING

Test case scheduling is an issue when testing complex and data-intensive DBAPs such as core banking systems. Restoring respectively loading a database snapshot of the complete system requires hours even for small or medium-sized banks. Let us assume that a regression test executes twenty test cases and loading a database snapshot takes five hours. If each test case requires loading a database snapshot, the loading alone needs 100 hours. Thus, reducing the number of database snapshot loads is a major issue.

Scheduling test cases to reduce the number of loads is broadly discussed and solved for black-box testing using one testing database snapshot respectively one database test state [16]. The problem is also relevant for white-box testing. White-box testing can benefit from semantically richer test case specifications. They allow additional optimizations. Also, one testing database state might not be sufficient. Banks charge their customers certain fees at the end of each quarter, other at the end of the year. To check whether the accounting takes place correctly, one has to check the balance sheet calculations at the end of a quarter and also after end-of-year processing. Two test case states are necessary.

Thus, following our philosophy of elaborating the broad range of challenges for DBAP testing, we describe a test case scheduling task for white-box testing by introducing the concept of a test case schedule and input correct test schedules while considering the semantics of white-box test cases before we formulate the optimization task. Therefore, we focus on shallow test cases, which rely especially on a coherent system with consistent data.

A formalization of test case scheduling requires the concept of a test schedule. A test schedule $\boldsymbol{S}$ is a sequence of actions. Each action either executes a test case or loads a database snapshot. All test cases are executed at least once.

Definition (Test Schedule): *Let $\mathcal{TC}=\{TC_1, TC_2, \ldots TC_n\}$ be the test case set and $\mathcal{L}=\{L_1, L_2, \ldots L_m\}$ be actions for loading database states $\Sigma_1, \Sigma_2, \ldots \Sigma_m$. Let $tc_i \rightsquigarrow a_j$ denote that action a_j represents the execution of a test case tc_i. Then a test schedule $\boldsymbol{S}$ is a pair $< \mathcal{A}, <_a >$ with the actions $\mathcal{A}$ to be performed and , $<_a$ defining the execution sequence with the following side conditions:* $\forall a \in \mathcal{A}: a \in \mathcal{TC} \cup \mathcal{L}$, $\forall tc \in \mathcal{TC}: \exists a \in \mathcal{A}: tc \rightsquigarrow a$.

If the database state before the execution of a test case is not the intended one (that is, it is not execution-path compliant), a different path is executed than the one intended. The concept of an input correct test schedule addresses this topic. Let TC_1 and TC_2 be two test cases with the constraint sets Σ_{I1}^*, Σ_{I1}^+, Σ_{O1} respectively Σ_{I2}^*, Σ_{I2}^+, and Σ_{O2}. We denote the state after the execution of TC_1 with Σ_A. Remember that Σ_{I1}^+, Σ_{I2}^+, and Σ_A are usually not explicitly specified constraints for complex DBAPs but are implicit definitions based on database snapshots.

Whether test case TC_2 can be executed without loading a database state depends on the relationship between Σ_A and Σ_{I2}^* respectively of Σ_A and Σ_{I2}^+.

- *Relationship Σ_A and Σ_{I2}^*:* Σ_A and Σ_{I2}^* are database states, i.e. implicitly defined constraint sets. Without loading the test state, we can only reason about the possibility of executing TC_2 after TC_1 without knowing the state after the execution of TC_1 for certain cases. If $\Sigma_{O1} \Rightarrow \Sigma_{I2}^*$ holds, we know we can

execute TC_2 directly after TC_1. If $\Sigma_{O1} \cup \Sigma_{I2}^*$ is inconsistent, we must not. We cannot decide a priori otherwise. However, the assumption is always that the execution of test case TC_1 succeeded.

- *Relationship Σ_A and Σ_{I2}^+:* Both sets are implicitly defined. We cannot reason anything a priori. After we have executed TC_2 *and* the test case succeeded *and* we know from a trace that the right path has been executed, then, and only then, Σ_A implies Σ_{I2}^+ (and even Σ_{I2}^*). Therefore, we write $\Re(\Sigma_{B(TC1)}, TC_1, TC_2)$ with $\Sigma_{B(TC1)}$ being the system state before the first test case has been executed. Certainly, information about $\Re$-conditions should be managed such that the optimization improves over time as described for black-box testing [16].

An input correct test ensures that test cases are executed with a correct database state, either by loading the corresponding database state before the execution or by ensuring that the execution took place under the $\Re$-condition. The $\Re$-condition can be known due to previous executions (learned and stored information). Otherwise, the condition can be checked only a-posteriori. Thus, the schedule might contain test case executions with inadequate prior states. However, at least one execution for each test case must be executed with a correct input state.

Definition (Input Correct Test Schedule): *Let $\mathcal{S}$ be a test schedule. Let $\mathcal{TC}$ be the set of test cases to be executed. Let a_{i-1} denote the action in S executed before a_i. Then we define the input correctness $\mathcal{I}(\mathcal{S})$ of schedule $\mathcal{S}$ as follows:*

$$\mathcal{I}(\mathcal{S}) \Leftrightarrow \forall tc \in \mathcal{TC}: \exists a_i \in \mathcal{A}: a_i \leftarrow / tc \wedge [(a_{i-1}=L_j \wedge L_j = \Sigma_{Ii}^+ \wedge L_j \Rightarrow \Sigma_{Ii}^*) \vee (\Re(\Sigma_{B(a_{i-1})}, a_{i-1}, tc))]$$

The optimization task is finding a schedule with minimal execution time for a given test case set. The optimal schedule can be found only if full $\Re$-condition knowledge exists. However, it is more important to find a suitable scheduling algorithm that is able to find a useful (but not necessarily perfect) schedule with less than complete knowledge and considering many database states and exploiting the semantics of white-box test cases.

9. DISCUSSION AND OUTLOOK

The sheer number of DBAPs on the market contrasts with the lack of tools for and basic understanding of testing DBAPs. Our contribution is to compile all relevant aspects of a complete testing process, including correctness criteria, test case types, and scheduling problems. Our process-oriented perspective helps practitioners identify at which steps they can improve their software testing process. Furthermore, they can develop an intuitive understanding of DBAP-specific testing risks such as data parallelism coverage and test data compliance. Researchers benefit from the terminology covering much of the currently highly fragmented research area.

The presented problems and challenges (often only solved with many restrictions) point out the questions currently unsolved. They might be a good starting point for future research. We see two main needs. Firstly, ready-to-use tools for automatically identifying shallow test cases would be a major improvement. Test cases should incorporate a corresponding database state as a complete database snapshot, but also as a constraint set. The constraint set could be used to check whether a given database state might be used as well. Secondly, deep test case identification is highly relevant, including a tool for generating varied test data. Even more useful would be visualizations: one for the relationship between terms in SQL statements and the result (changed/unchanged by a statement or selected by the query or not) and one of how two statement versions influence the rows affected by the statement. Thereby, the emerging field of testing DBAPs could influence and improve mainstream software development.

Acknowledgements. The author would like to thank Michael Mlivoncic for the valuable discussions.

REFERENCES

[1] Haller, K.: Data Migration Project Management and Standard Software – Experiences in Avaloq Implementation Projects, *DW 2008 Conference, Lecture Notes in Informatics,* St. Gallen, Switzerland, 2008

[2] Haller, K.: Towards the Industrialization of Data Migration: Concepts and Patterns for Standard Software Implementation Projects, *21st Int. Conf. on Advanced Information Systems (CAiSE'09)*, 2009, Amsterdam, The Netherlands

[3] Zhu, H., Hall, P. May, J.: Software Unit Test Coverage and Adequacy, *ACM Computing Surveys*, Vol. 29 (4), 1997

[4] Graham, D., et al.: Foundations of Software Testing, Thomson, 2008, London, UK

[5] http://www.ibm.com/software/awdtools/purifyplus/

[6] Willmor, D. and Embury, S.: Exploring Test Adequacy for Database Systems, *3rd UK Software Testing Research Workshop (UKTest)*, 2005, Sheffield, UK

[7] Suárez-Cabal, M. J. and Tuya, J.: Using an SQL coverage measurement for testing database applications, *SIGSOFT*, 2005, Newport Beach, CA

[8] Grant, J.: Null values in SQL, *SIGMOD Record*, Vol. 37 (3), September 2008

[9] Chays, D., et al.: An AGENDA for testing relational database applications, *Software Testing, Verification, and Reliability, Vol. 14(1) , 2004*

[10] Willmor, D. and Embury, S.: An Intensional Approach to the Specification of Test Cases for Database Systems, *28th Int. Conf. on Software Engineering*, 2006, Shanghai, China

[11] Deng, Y., Frankl, Ph., Chays, D.: Testing Database Transactions with AGENDA, *27th International Conference on Software Engineering (ICSE),* 2005, St. Louis, MO

[12] Binning, C., Kossmann, D., Lo, E.: Towards Automatic Test Database Generation, *IEEE Bulletin on Data Engineering,* Vol. 31(1), 2008

[13] Binning, C., et al.: MultiRQP – Generating Test Databases for the Functional Testing of OLTP Applications, *1st International Workshop on Testing Database Systems*, 2008, Vancouver, Canada

[14] Chays, D., et al.: A Framework for Testing Database Applications, *International Symposium on Software Testing and Analysis (ISSTA)*, 2000, Portland, OR

[15] Willmor, D. and Embury, S.: A regression test selection technique for database-driven applications, *21st Int. Conf. on Software Maintenance (ICSM),* 2005, Budapest, Hungary

[16] Haftmann, F., Kossman, D., Lo, W.: A Framework for Efficient Regression Tests on Database Applications, *VLDB Journal*, Vol. 16 (1), 2007