An Approach to Perform Automated Functional Testing in Database-Driven Applications

Andreza M. F. V. de Castro[1], Awdren Fontão[1], Arilo C. Dias-Neto[2]

[1] Instituto Nokia de Tecnologia – INdT, Manaus, AM
andreza.dy@gmail.com ; awdren.fontao@indt.org.br

[2] Experimentation and Software Testing Group– ExperTS, UFAM, Manaus, AM, Brazil
arilo@icomp.ufam.edu.br

Abstract. This paper presents an approach to performing automated functional testing in database-driven application. The goal is to provide test scripts that check contents in databases after CRUD (create, retrieval, update, and delete) operations using a software application. Based on automation tools already developed that test graphical user interface (GUI) elements, new assert functions were proposed and implemented to check data in databases. An instantiation of the proposed approach is described extending the Selenium RC tool to perform automated tests in web applications checking data stored in the database. A case study is described to analyze the impact of the proposed approach in terms of effort and rate of automation in the testing for a database-driven application. The results suggest a significant reduction (91% and 88%) in the effort to execute automated tests in database when compared to, respectively, manual and semi-automated execution.

Keywords: Software Testing, GUI Testing, Test Automation, Database Testing.

1 Introduction

Database-driven applications play a central role in our information based society. They are the core of most modern software systems, managing and manipulating data related to a specific domain. Thus, different operations (e.g.: CRUD operations – create, retrieval, update, and delete) can be performed on these data, and their success represents one of the main indicator of quality in these applications [1].

Testing whether these applications behave correctly is of great importance. Because of the huge space of possible database states that must be considered, these applications pose new challenges to software testing, such as to confirm if the results presented in the software's graphical user interface (GUI) after an operation is in accordance with the results processed by the database [2]. For instance, if the software's GUI presents a message of success after adding a new entity, we believe these data were fully stored in the database. However, a failure can happen during this operation without being displayed in the software interface. In this scenario, the differences of states/contents in a database among different operations may help to identify potential failures in software [3]. However, the manual verification of these differences is a hard and time-consuming activity. Thus, automated testing [1] would be an interesting solution to perform functional testing in database-driven applications quickly and efficiently.

The definition for Software Test Automation means to automate software testing activities including the development and execution of test scripts, verification of

testing requirements, and the use of automated testing tools [4]. The use of this practice may reduce the effort to perform testing in a software project, that is, we will be able to execute a highest number of test cases in less time [5]. Contextualizing the automation strategies for software testing in database-driven applications, we can observe several strategies proposed to support testing of SQL scripts written for a software or test data generation for SQL queries [2], which are important in the software development. However, these strategies must be executed during the development phase and, therefore, they cannot assure that the database operations are being correctly performed when the software is deployed to its final users.

Thus, integrating the database testing into a functional testing strategy (e.g.: GUI testing [6]), which is usually adopted in software projects, could contribute to reduce the effort and impact of testing in software projects. It could also minimize the risk of software deployment with failures in database operations, because this testing would be also performed in the last stage before the software deployment: during the functional testing execution.

This paper presents an approach to support automated testing that integrates functional GUI testing and data checking in database-driven applications. It could be applied for database-driven applications developed in different software platforms. In this work, we instantiate this approach to test web-based applications extending a functional GUI testing tool very popular in the software industry: Selenium RC tool [7]. This extension, called *SeleniumDB*, implements new functions in the Selenium RC's core to test automatically web-based applications with asserts that check data in database's tables. *SeleniumDB* was used in a case study and the results indicated a significant reduction (91% and 88%) in the effort to execute test cases using the proposed tool when compared to, respectively, manual and semi-automated strategies.

Besides this introduction, this paper is organized as follows: Section 2 discusses some challenges and related Works in testing database-driven applications. Section 3 presents the approach proposed to support automated database testing. Section 4 describes the extension of the Selenium RC tool developed to instantiate the proposed approach. Section 5 described a case study performed to apply the tool in a testing benchmark and the comparison between three strategies (manual, semi-automated, and automated tests execution). Section 6 presents the conclusions and future work.

2 Testing in Database-Driven Applications

The process of testing software is difficult because it requires the description of all aspects involved in a software system and the adequate testing of them. Device drivers, operating systems, and databases are all aspects of a software system's environment that are often ignored during testing [8]. According to Kapfhammer and Soffa [9], in the last years, a wide range of traditional software testing techniques have been proposed, implemented, and evaluated. However, relatively little research has specifically focused on the testing and analysis of database-driven applications.

Intuitively, a database-driven application is a program whose environment always contains one or more databases. Given the preponderance of high quality database management systems and the number of organizations that are now collecting an unprecedented amount of data, there is a clear need for software testing techniques that test an application and its interaction with a database [9].

2.1 Challenges in Testing Database-Driven Applications

We can find in the technical literature several challenges to perform testing in database-driven applications, such as:

- **Challenge 1)** Even though the state space of an application that interacts with a relational database is well-structured (because it is described by a relational schema), it is also essentially infinite, making hard the testing tasks [2].
- **Challenge 2)** The representation of a database-driven system must clearly indicate the coupling points between the program and the entities within the database to contribute to the testing activities [10].
- **Challenge 3)** Test adequacy criteria for database-driven applications must ensure the existence of tests that can reveal the types of faults that are commonly found in programs that interact with databases [9].
- **Challenge 4)** Selecting interesting database states and populating the database with meaningful values are important tasks for the success of testing in database-driven applications [2].
- **Challenge 5)** At checking the results, testers must pay attention to the state of the database after running the application [2].
- **Challenge 6)** In data generation for database testing, deciding whether the database will be populated with live or synthetic data is an important decision. The live data may not reflect a sufficiently wide variety of possible situations that could occur. Even if the live data encompasses a rich variety of interesting situations, it might be difficult to find them, especially in a large database, and difficult to identify appropriate user inputs to exercise them and to determine appropriate outputs [2].
- **Challenge 7)** Generating synthetic data for database testing is not trivial. Since the database state is a collection of relation states, the generated data cannot ignore an important aspect of the database schema: the integrity constraints. We do not want to populate the database with just any tables, but rather, with tables that contain both valid and interesting data [2].
- **Challenge 8)** Database-driven applications can cause the violation of database integrity in different ways, and the testing strategy must deal with all of them [11]. A program can violate the validity or completeness of its database if insertion/edition/deletion of records into a database that does not reflect the changes in the real world are performed or vice-versa.

Some approaches are available in the technical literature to deal with some of these challenges, as described in the next subsection. In this work, we deal with some of these challenges: Challenges 5, 6, and 8.

2.2 Related Works

Several works have been proposed in the technical literature to support testing in database-driven applications. They are related to the challenges described in Section 2.1. Regarding query-based test cases generation, in [14] and [15], it is presented techniques that build or modify a database for each test case intentionally, in which the user provides a query that specifies the pre- and post-conditions for the test case. These techniques depend on the user-defined data generator functions as opposed to a set of values partitioned into groups that are meaningful to the user and easier to maintain. Moreover, in [12] and [13], symbolic execution was used to generate input and database states that will satisfy certain coverage requirements or certain requirements on the intermediate or final results of queries.

Analyzing works related to test adequacy criteria for database-driven applications, Kapfhammer and Soffa [9] propose a family of test adequacy criteria that can be used to assess the quality of test suites for database-driven applications. Their test adequacy criteria use dataflow information that is associated with the entities in a

relational database. Furthermore, they developed a unique representation of a database-driven application that facilitates the enumeration of database interaction associations. In another work, Halfond and Orso [16] introduce a test adequacy criterion that is based on coverage of the database commands generated by an application and specifically focuses on the application-database interactions.

Chays et al. [2] describe a tool that populates databases with data that satisfies the schema constraints in a fashion that is reminiscent of the category-partition method. This approach measures adequacy with respect to the specification of the database and the program and additional testing heuristics. It always requires tester intervention. Zhang et al. [17] describe an approach that attempt to generate database states that are consistent with respect to the constraints in the relational schema. Yet, this approach does not explicitly include a test adequacy criterion designed to isolate defects in the programs that interact with databases. While Dauo et al. [18] propose a regression testing technique for database-driven applications, their exploration of data flow issues does not include a discussion of the representation of a database-driven application or a formal understanding of a database interaction association.

Some tools have also been proposed in the technical literature, such as: the SIKOSA Project [19], a capture-and-replay tool for functional regression testing in database-based applications, the DART tool [3], a tool to support regression testing in database-based applications, DBUnit [20], a JUnit framework extension that can be used to perform unit tests using database operations, SWAT [21], a test tool that allows users to automate web application testing in multiple browsers, and the Apollo tool [22], is a supporting technique to adequately test DB applications (in particular, interactions application-DB).

In general, these approaches and tools support testing in database-driven applications. However, none of them supports automated testing that integrates functional GUI testing (testing based on GUI elements provided by a tool) and data checking in database-driven applications, what is the goal of this work. Therefore, our proposal could be applied together to the related works presented in this section. In the next section we present the approach developed in this work to support automated functional database testing.

3 Approach for Automated Database Testing

Considering the challenges presented in section 2, our proposed approach aims to support automated testing in database-driven applications already developed and ready to be functionality tested. This approach consists in the integration of automated database testing with automated functional testing based on the system's GUI (called GUI Testing). In GUI Testing, the application is analyzed in the user's perspective. Test cases are designed considering the system as a black box, hiding the coding details, based on GUI elements that are used as input to test cases generation [6].

The proposed approach focuses on one category of operations in databases from SQL queries: operations manipulated by DML (Data Manipulating Language – statements used for managing data within schema objects). This language (category of SQL queries) provides operations that change the content of database's elements, such as insertion (INSERT) of new records into database's tables, edition of a table's record from some condition (UPDATE), deletion of a table's records from a condition (DELETE), and retrieval of records from tables (SELECT). These operations are called CRUD: create, retrieval, update, and delete.

Thus, some scenarios were mapped considering the database's state before and after one operation using a database-driven system:

1) A known record is inserted into a table

A new record is inserted into a table and we have enough information to identify uniquely it, such as its primary key (PK). Thus, after this insert operation we need to confirm if the inserted record can be found in the changed table. This checking can be done selecting in a table the record that contains the inputted PK. The expected result of this checking cannot be null.

2) An unknown record is inserted into a table

A new record is inserted into a table, but we do not have enough information to identify uniquely it, such as its PK. Thus, we need to confirm if the database state before and after this insert operation changed just including this new unknown record. This checking can be done in several database management systems (DBMS) selecting the last record inserted into a table (it must be the new inserted one).

3) A known record is changed in a table

A record is updated in a table. Thus, after this operation we need to confirm if the updated record can be found in the table and all changed columns were updated in the database's table. This checking can be done in two ways: selecting in a table the updated record and confirming (1) if all columns were updated correctly with the new data and (2) if all the old data are not present in the updated columns.

4) A known record is removed from a table

A record is deleted from a table. Thus, after this operation we need to confirm if the removed record cannot be found in the table. This checking can be done selecting a record that contains the deleted PK. The expected result of this checking must be null.

5) A set of records should be filtered from table(s)

A filter is performed in the application and we need to confirm if the result returned by the application contains all records that attend the filter's condition. In this scenario, SQL queries allow different possibilities of filter, such as single select (selecting from a unique table), combined selection (by JOIN or MERGE operations), or grouping data from different sources. This checking can be done confirming that the SQL query executed by the application returns all records in the database that attend the indicated condition. Checking this operation would be more complex when more than one data source is involved (e.g.: join/merge between two tables).

The Figure 1 presents the steps required to perform database testing considering the five scenario presented above. These steps can be done following two different strategies: manual and automated checking of database's content.

In a (traditional) manual database testing strategy, firstly, the tester is responsible for creating functional test scripts in any test automation tool to evaluate GUI elements (step 1 in Figure 1). Then, the selected tool executes the created test scripts (step 2). Finally, the tester must manually access the database to check if all operation in the database were correctly performed (step 3).

In the proposed database testing approach, the tester also needs to create test scripts to evaluate GUI elements (step 1). However, in this proposal they will be also able to create database checking scripts to confirm if the operations in the database were correctly performed (step 4). Then, the automation tool executes the created test scripts to check GUI elements (step 2) and also the database operations' correctness (step 5). Thus, the step 3 (manual database's checking) is eliminated of the functional testing strategy, because this checking is performed automatically by the test tool.

Thus, to instantiate the proposed approach we need to extend a tool that supports the creation of GUI test scripts implementing new assert functions to check content in database's tables. There are several commercial and open source tools available to automate GUI testing, such as Watir, Canoo WebTest, JMeter, Selenium, amongst

others. Therefore, we believe it is not necessary to develop a new tool to implement the proposed approach. We could adapt the tools already developed to test different platforms (e.g.: desktop, mobile, web-based, embedded software) that are available in the technical literature or the software industry.

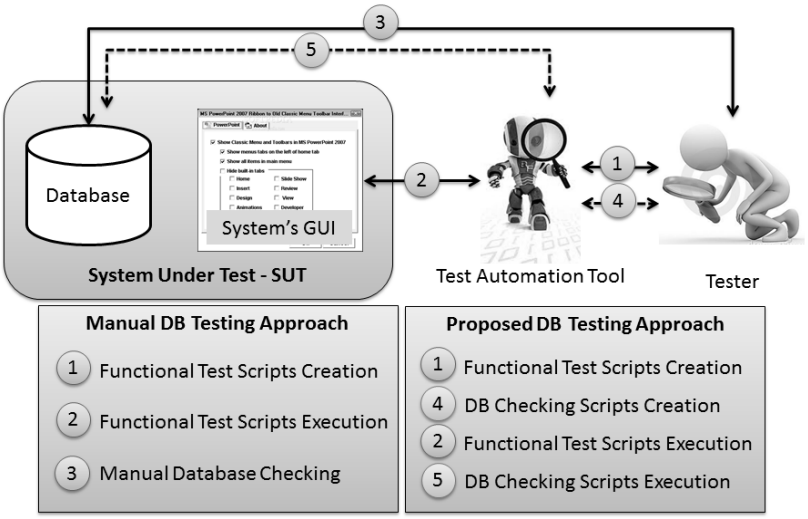

Figure 1. Steps to Perform Manual and Automated Database Testing.

In the next section, we present the instantiation of the proposed approach to support automated database testing for web-based applications extending the Selenium framework [7], which is very popular in the software industry to support GUI testing in web-based applications.

4 Instantiating the Proposed Database Testing Approach for Web-Based Applications

This section presents the instantiation of the proposed approach to support automated database testing. It consists in the extension of the Selenium RC tool to perform tests in web-based applications that require checking data in databases. This verification is performed using new functions implemented into the Selenium framework's core. These functions allow opening and closing database connection and comparing test data with data stored in the database used by the web-based application following the five scenarios presented in Section 3.

This instantiation of automated database testing for web-based applications has been briefly introduced and evaluated in a real software project in [23]. This solution aims to contribute to the system quality by reducing the effort during the testing process, since the verification of GUI and database elements will be performed automatically at the same time during execution of the test scripts.

The creation of new assert functions was feasible because of the infrastructure provided by the Selenium framework, which supports extensions using different programming languages. Due to the limitation in the Selenium framework to support automated testing in web-based applications that access databases, we extended the Selenium RC tool to deal with asserts that query data in these databases.

This extension of the Selenium RC tool (we called it *SeleniumDB*) was developed using Java, so the test cases can be executed using the JUnit framework [24]. This

technology was selected because it is very popular technology in the software engineering community and the Selenium RC tool's source code is available in Java.

Regarding the choice of database management system (DBMS), *SeleniumDB* would be able to connect to any DBMS. In this case, we need to create a specific function to perform the connection to the database. In the first moment we decided to implement the connection function for two of the most popular DBMS's: MySQL and PostgreSQL. These functions must receive some parameters (server, username, password, port number, and database name) to perform the connection by JDBC (Java API that supports connection to databases). Moreover, we need to create a function to disconnect the database. In this case, only one function is necessary, because it just receives the object connection and finalizes it. The signatures of these three functions are listed in Figure 2.

```
public Connection connectionMySQLDB(String username, String
   password,String server,String portnumber, String dbName) throws
   SQLException;

public Connection connectionPostgreSQLDB(String username,String
   password,String server,String portnumber,String dbName) throws
   SQLException;

public void closeConnectionDB(Connection con) throws SQLException;
```

Figure 2. Signature of the database connection/disconnection methods.

To support automated testing with databases, six new assert functions were implemented in Selenium RC's core. These functions implement the approach proposed in Section 3. They compare data in tables' columns with texts submitted by testers in test scripts or texts filled in GUI elements of the web-based application under test. The implemented assert functions attend to all five scenarios mapped in the Section 3. However, the scenario 5 (*A set of records should be filtered from tables*) is partially attended by this instantiation, because only single select (selecting from a unique table) is supported in this version.

From now, each assert method and their role in automating database's data check is presented.

4.1 *assertKeyDBPresent(con, table, column, data)*

This function searches for a value that represents a table's primary key in a specific column (Figure 3). When the test datum is found in the database's table, the *assertTrue* method is called (test case passes). Otherwise, the fail method is called with an error message (test case fails).

```
public void assertKeyDBPresent(Connection con,String table, String
column,String data){
Statement stmt = null;
try{
  stmt = con.createStatement();
  String q = "";
  q += "Select * from "+table+" WHERE "+column+" = '"+data+"'";
  ResultSet rs = stmt.executeQuery(q);
  if(rs.next()){        rs.getString(column);    assertTrue(true); }
  else {                fail("Test Failed!");}
}
catch (SQLException e){ e.printStackTrace();}
}
```

Figure 3. Extract of the *assertKeyDBPresent* function's source code [23].

4.2 *assertKeyDBNotPresent(con, table, column, data)*

The inverse of the previous function, this one checks if a test datum does not exist in the table's primary key column. Its implementation is very similar to the previous function, however when the test datum is found, the fail method is called (test case fails). Otherwise, the *assertTrue* method is called (test case passes).

This assert function attends the scenario 4 in the proposed approach (Section 3). It checks if a deleted record (identified by its primary key inputted as parameter) was really removed from a specific table.

This assert function attends the scenario 1 in the proposed approach (Section 3). It checks if a known record (identified by its primary key inputted as parameter) exists in a specific table, returning true if it is found and false, otherwise.

4.3 *assertColumnDBPresent(con,table,keycolumn,keydata, column, data)*

Function developed to search for a test datum in a column that does not represent a table's primary key. In this case, we cannot check the test data only in the indicated column. As it accepts duplicated data, the test datum could be in more than one record, which could affect the test results. Therefore, the function searches for an identifier submitted as parameter in the table's primary key column and then checks the existence of the test datum in the desired column.

Its implementation is very similar to the function *assertKeyDBPresent*. However, the query used to search the test datum in the database is a little different (Figure 4).

```
public void assertColumnDBPresent(Connection con,String table,
String keycolumn,String keydata, String column, String data){
  Statement stmt = null;
  try{
   stmt = con.createStatement();
   String q = "";
   q += "Select * from "+table+" WHERE "+keycolumn+" = '"+keydata+"'
AND "+column+" = '"+data+"'";
   ResultSet rs = stmt.executeQuery(q);
   if(rs.next()){       rs.getString(column);  assertTrue(true);  }
   else {               fail("Test Failed!"); }
  }
  catch (SQLException e){       e.printStackTrace();  }
}
```

Figure 4. SQL Query used in function *assertColumnDBPresent* [23].

This assert function attends the scenario 3 and 5 in the proposed approach. It checks if the content of a known record matches with a text inputted as parameter. Thus, it could be used to confirm the success of update and select operations in database's tables.

4.4 *assertColumnDBNotPresent(con, table, keycolumn, keydata, column, data)*

The opposite of the previous function, this one checks if a test datum does not exist in a table's column. Its implementation is very similar to the previous function. However, when the test datum is found, the fail method is called (test case fails). Otherwise, the *assertTrue* method is called (test case passes).

This assert function also attends the scenario 3 (Section 3) in the proposed approach. It could be used to check if old data were changed in the update record.

4.5 *assertLastRecordDBPresent(con, table, idcolumn, column, data)*

This function searches for a datum in the last inserted record in a database's table column other than the primary key. This function should be appropriate when we do not have access to the primary key that identifies uniquely the last record stored in a table (e.g.: insertion operation in a database's table) and the table has an id column. When the test datum is found in the database's table, the *assertTrue* method is called (test case passes). Otherwise, the fail method is called (test case fails). Figure 5 presents the code used in this function to get the last record in a database.

This assert function attends the scenario 2 in the proposed approach (Section 3). It checks if an unknown record (*the last recorded inserted into a table*) contains the datum inputted as parameters in this function, returning true if it is found and false, otherwise.

```
public void assertLastRecordDBPresent(Connection con,String
table,String idColumn,String column,String data){
  ResultSet rs = null;
  PreparedStatement pstmt = null;
  try{
   String q ="";
   q += "Select * from "+table+" ORDER BY "+idColumn+" DESC LIMIT 1";
   pstmt = con.prepareStatement(q);
   rs = pstmt.executeQuery();

   if(rs.next()){ String columnData = rs.getString(column);
    if(data.equalsIgnoreCase(columnData.trim())){ assertTrue(true);}
    else{ fail("Test Failed!"); }
   }else { fail("Test Failed!");}
  }
  catch (SQLException e){  e.printStackTrace(); }
 }
```

Figure 5. Extract of the *assertLastRecordDBPresent* function's code [23].

4.6 *assertLastRecordDBNotPresent(con,table,column,data)*

The inverse of the previous function, this one checks if a test datum does not exist in the last inserted record in a database's table column. Its implementation is very similar to the previous function, except that when the test datum is found, the fail method is called (test case fails). Otherwise, the *assertTrue* method is called (test case passes).

This assert function also attends the scenario 4 in the proposed approach (Section 3). It checks if an unknown record (the last recorded inserted into a table) does not contain the datum inputted as parameters in this function. This function could be used to confirm that the last record in a table was deleted correctly (e.g.: in a UNDO operation after an insertion using the application).

Thus, the new implemented assert functions attends all scenarios mapped in the approach proposed in Section 3.

It is possible to access an example of test scripts developed to be used with *SeleniumDB* and the source code implemented in this tool in the URL www.icomp.ufam.edu.br/experts/seleniumdb_project.zip. More information regarding this extension are published in [23]. In the next section will be presented a case study describing the use of this tool in a web-based application.

5 Case Study

5.1 Case Study Goals and Design

We conducted a case study to investigate the effectiveness of the tool developed in this work (*SeleniumDB*) to instantiate the approach proposed to support automated database-driven applications. For the purposes of this study, we selected a database-driven web benchmark used in previous works to support the evaluation of database testing approaches. Next, we selected some functionalities of this benchmark and we designed a functional test suite to verify them.

In order to evaluate the effectiveness of the developed tool, we measured the time required to execute all test cases available in the designed test suite. As the same test cases were executed using all approaches, we did not measure the time required to create the test suite nor the number of failures detected by it.

Three strategies of test execution were evaluated: manual, semi-automated and automated. In the manual execution, the tester should open the browser, login into the system, do the action indicated in the test case (e.g.: register customer, search products, pay order, etc.), open the database tool and access the database table to check the data. In the semi-automated execution, some steps were done by the test script implemented using Selenium RC (open the browser, login into the system, do the action indicated in the test case) and other steps were done manually by the tester (open the database tool and access the database table to check the data). Finally, in the automated execution, all steps were performed by the test scripts built by the testers using the *SeleniumDB*, including the access to the database's table to check the test data.

The time required to execute manually test cases was measured by both researchers, simulating the behavior of testers using the application and the support tools. In the automated tasks, Selenium RC and *SeleniumDB* reported the execution time.

After that, we analyze the results obtained from all testing approaches to indicate which one would be more effective to support database-driven application functional testing.

5.2 Case Study Benchmark: TPC-W

The extension of Selenium RC (*SeleniumDB*) that instantiates the approach proposed to support automated database-driven applications was evaluated with a database-driven web benchmark called TPC-W [25]. This benchmark defined the complete Web-based shop for searching, browsing and ordering books. TPC-W standard describes all pages that must be present in the shop (including sample HTML code) and database schema, that will be used in this case study. This benchmark was selected for this case study since it has been used in previous works to support the evaluation of database testing approaches [1][2].

In this case study, we focused on 2 functionalities provided by TPC-W benchmark: *Customer Registration and Admin Request*.

- **Customer Registration:** functionality that allows a User to provide the information necessary to register as a known Customer (inputting his/her username and password) or as a new Customer (inputting several information, such as name, address, phone, email, and birth date) and to submit their registration.
- **Admin Request:** functionality that allows a user with Admin's profile to request the update some information (price, image, and thumbnail) of an item (book) in the store.

5.3 Test Cases Designed for TPC-W

The test suite was designed for both functionalities using the category partition method [26]. The business rules related to each functionality is described in the TPC-W benchmark specification [25].

The list of designed test cases can be found in Table 1, where the results of their execution are described. This analysis is discussed in the next subsection.

Table 1. Results of Test Cases Execution (in seconds)

Test Cases	Manual Execution's Time			Semi-automated Execution's Time			Automated Execution's Time	
	Functional Test	DB Check	Total	Functional Test	DB Check	Total	Functional and DB Test	Total
CR-1: Known Customer (success)	14.0	47.0	61.0	7.4	47.0	54.4	8.0	8.0
CR-2: Known Customer (invalid username)	18.0	46.0	64.0	7.8	46.0	53.8	8.4	8.4
CR-3: Known Customer (invalid password)	19.0	49.0	68.0	7.3	49.0	56.3	7.8	7.8
CR-4: Unknown Customer (success)	77.0	155.0	232.0	10.0	155.0	165.0	8.3	8.3
CR-5: Unknown Customer (repeated name)	75.0	80.0	155.0	9.9	80.0	89.9	8.2	8.2
CR-6: Unknown Customer (invalid email)	70.0	91.0	161.0	9.9	91.0	100.9	8.3	8.3
CR-7: Unknown Customer (fields in black)	57.0	60.0	117.0	7.6	60.0	67.6	8.2	8.2
AR-1: Edit Item (success)	35.0	110.0	145.0	13.2	110.0	123.2	12.9	12.9
AR-2: Edit Item (field not filled out)	17.0	76.0	93.0	7.7	76.0	83.7	11.7	11.7
AR-3: Edit Item (invalid negative price)	28.0	119.0	147.0	12.8	119.0	131.8	13.8	13.8
AR-4: Edit Item (invalid price = 0)	26.0	60.0	86.0	12.6	60.0	72.6	13.5	13.5
AR-5: Edit Item (invalid picture)	27.0	55.0	82.0	13.4	55.0	68.4	14.6	14.6
AR-6: Edit Item (invalid thumbnail)	29.0	50.0	79.0	13.5	50.0	63.5	13.9	13.9
TOTAL	**492.0**	**998.0**	**1490.0**	**133.0**	**998.0**	**1131.0**	**137.4**	**137.4**

5.4 Results Analysis

Using both strategies, 13 test cases were executed. The results are presented in Table 1. The total time spent for manual tests execution was 1490 seconds (around 25 minutes) and 67% of this time (998 seconds) was spent checking the test data in the database. The time spent for test cases execution using the semi-automated strategy was 1131 seconds (around 19 minutes) and 88% of this time (again 998 seconds) was spent checking the test data in the database. This result indicates a reduction of 24% in the final time spent for test cases execution using a semi-automated functional testing tool when compared to manual tests execution.

Analyzing the use of *SeleniumDB* (automated functional and database testing), it was possible to observe that the total time spent for test execution was 137.4 seconds (around 2.5 minutes). In this strategy, the checking of test data in the database is performed in the same script used for functional test. Thus, they cannot be divided into two steps. This result indicates a reduction of 88% in the final time spent for test

cases execution when compared to the semi-automated test execution and a reduction of 91% in the final time spent for test cases execution when compared to manual tests execution.

Moreover, using *SeleniumDB*, the automation rate of test cases designed for these two functionalities was 100% (13 automated test cases ÷ 13 designed test cases). It was not necessary to perform any manual step to execute these test cases.

The analyzed results do not consider the effort to create the test scripts to be executed by the automation tools. If considered, this time could increase the time required to use the semi- and automated strategies. On the other hand, we are not considering the saved effort using the semi- and automated strategies when perming regression testing (re-execution of all test cases for a new software version). The effort to create automated test scripts happens only once in a software project. Thus, in regression testing they can be executed automatically without additional efforts. Therefore, this effort is dissolved among the iterations. When performing manual regression testing, we do not have additional effort to create automated test scripts, but the effort to execute all test cases will be the same with all software versions.

This case study is not conclusive. It provides just insights of the feasibility and effectiveness of the proposed approach in supporting automated database testing using a web-based application. New studies need to be performed in other software platforms and extending other automation tools to increase the confidence in the results.

As a start point, the extension of Selenium RC (*SeleniumDB*) that instantiates the approach proposed to support automated testing in database-driven applications was also evaluated in a real software project developed by a Brazilian company [23]. The results indicated a reduction of 88% and 92% in the effort to execute a test case suite when compared, respectively, to semi-automated and manual tests execution strategies. We can observe the results were very similar to the results obtained in the case study described in this work (88% and 91%), suggesting a real reduction in the effort to execute test cases using the proposed approach. Details of this study performed in the software industry are presented in [23].

6 Conclusions

Automated Database Testing can be used to support software engineers in the evaluation of database-based web applications, reducing the effort to perform this task when compared to manual or semi-automated strategies. Some problems in database testing would be the use of other tool to automate specific tests and the effort to create additional test scripts to deal with this aspect.

In order to solve these problems, this paper proposes an approach to support automated database-driven application testing integrated to functional GUI testing approaches. This approach was instantiated to the scenario of web-based application testing. Thus, we extended a very popular functional GUI testing tool (Selenium RC) to implement the proposed approach, supporting database testing in web-based applications. We called this solution *SeleniumDB*.

Assert functions suggested by the proposed approach to support automated database-driven application testing were implemented into the Selenium framework's core to deal with test data checking in databases. These functions are applied to different scenarios where data need to be checked in databases, such as: searching for

known and unknown data, inserting, deleting, or updating records in tables.

The tool proposed was evaluated by applying it in case study using a database testing benchmark and comparing the obtained results to other two strategies (manual and semi-automated tests execution) regarding the effort (time) required to execute a test case suite. The results indicated a reduction of 88% and 91% in the effort to execute a test case suite when compared, respectively, to semi-automated and manual tests execution strategies.

As future works, we intend to instantiate the proposed approach in other software platforms, such as embedded and desktop systems. As another extension, we will implement new assert functions to provide different scenarios not covered by the actual solution regarding tests in a database including, for instance, assert functions that perform a join among different tables in a database. Finally, we intend to apply the proposed approach in other scenarios in the software industry to evaluate its feasibility, effectiveness, and efficiency in this context.

ACKNOWLEDGMENT

The authors would like to thank FAPEAM, CNPq (process 575696/2008-7) and INCT-SEC (processes 573963/2008-8 and 08/57870-9) for supporting this research.

REFERENCES

[1] Y. Deng, P. Frankl, D. Chays (2005), “Testing database transactions with AGENDA”. In *Proceedings of the 27th international conference on Software engineering* (ICSE '05). ACM, New York, NY, USA, 78-87.

[2] D. Chays, S. Dan, P. Frankl, F. Vokolos, E. Weber (2000); “A framework for testing database applications”. In Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00), Mary Jean Harold (Ed.). ACM, NY, USA, 147-157.

[3] R. Rogstad, L. Briand, R. Dalberg, M. Rynning, E. Arisholm (2011); "Industrial Experiences with Automated Regression Testing of a Legacy Database Application"; In: 27th IEEE International Conference on Software Maintenance (ICSM), pp. 362-371..

[4] E. Dustin, J. Rashka, J. Paul (2008) “Automated Software Testing: Introduction, Management, and Performance”. Addison-Wesley Professional.

[5] K. Karhu, T. Repo, O. Taipale, K. Smolander (2009), "Empirical Observations on Software Test Automation," In: International Conference on Software Testing Verification and Validation (ICST '09), pp. 201-209.

[6] B. Mu, M. Zhan, L. Hu (2009), “Design and Implementation of GUI Automated Testing Framework Based on XML”, In: *WRI World Congress on Software Engineering - Volume 04* (WCSE '09), Vol. 4. IEEE Computer Society, Washington, DC, pp. 194-199..

[7] Introduction - Selenium Documentation. Available at: http://seleniumhq.org/. Access in: April, 17th 2013.

[8] J. A. Whittaker (2000), “What is software testing? And why is it so hard?”, IEEE Software, 17(1):70–76, January/February 2000.

[9] G. M. Kapfhammer and M. L. Soffa (2003), “A family of test adequacy criteria for database-driven applications”. In *9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (ESEC/FSE-11). ACM, New York, NY, USA, 98-107.

[10] Z. Jin and A. J. Offutt (1998(, “Coupling-based criteria for integration testing”. Software Testing, Verification & Reliability, 8(3), pp. 133–154.

[11] A. Motro (1989), “Integrity = validity + completeness”. ACM Transactions on Database Systems, 14(4), pp. 480–502.

[12] C. Binnig, D. Kossmann, E. Lo, and M. T. Ä Ozsu (2007), “Qagen: generating query-aware test databases”. In SIGMOD '07: 2007 ACM SIGMOD international conference on Management of data, pp. 341-352, New York, NY, USA, ACM.

[13] M. Emmi, R. Majumdar, and K. Sen (2007), “Dynamic test input generation for database applications”. In ISSTA '07: 2007 International Symposium on Software Testing and Analysis, pp. 151-162, New York, NY, USA, ACM.

[14] D. Willmor and S. M. Embury (2006), “An intensional approach to the specification of test cases for database applications”. In ICSE '06: Proceedings of the 28th international conference on Software engineering, pp. 102-111, New York, NY, USA, ACM.

[15] D. Chays, J. Shahid, and P. Frankl (2008), “Query-based test generation for database applications”. In *Proceedings of the 1st international workshop on testing database systems* (DBTest '08). ACM, New York, NY, USA, Article 6, 6 pages.

[16] W. Halfond, A. Orso (2006); “Command-Form Coverage for Testing Database Applications”. In: Automated Software Engineering, Vol. 16, pp.201-209.

[17] J. Zhang, C. Xu, and S.C Cheung (2001) “Automatic generation of database instances for whitebox testing”. In Proceedings of the 25th Annual International Computer Software and Applications Conference, pp. 161-165, October 2001.

[18] B. Daou, R. Haraty, and N. Mansour (2001), “Regression testing of database applications”. In Proceedings of the 2001 ACM Symposium on Applied Computing, pp. 285–289. ACM Press.

[19] F. Haftmann, D. Kossmann, E. Lo (2007); “A framework for efficient regression tests on database applications”. In: The International Journal on Very Large Data Bases, Vol. 16, (January2007), pp. 145-164.

[20] ABOUT DBUNIT. Available at: http://www.dbunit.org. Access in: April, 17th 2013.

[21] N. Alshahwan, M. Harman (2011); “Automated web application testing using search based software engineering”. In: Automated Software Engineering (ASE), pp. 3-12.

[22] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M. Ernst (2010); “Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking”. Journals & Magazines, Vol. 36, pp. 474-494.

[23] A. Castro, G. Macedo, E. Collins, A. Dias-Neto (2013), “Extension of Selenium RC Tool to Perform Automated Testing with Databases in Web Applications”, In: Workshop on Automated Software Test (AST 2013), San Francisco, USA.

[24] K. Beck and E. Gamma (1998), “Test infected: Programmers love writing tests”. Java Report, 3(7), July 1998.

[25] Transaction Processing Performance Council. TPC-C. http://www.tpc.org, 2013.

[26] T. J. Ostrand and M. J. Balcer (1988), “The category-partition method for specifying and generating functional tests”. Commun. ACM, 31(6), pp. 676-686.