International Journal of Advanced Science and Technology
Vol.109 (2017), pp.53-66
http://dx.doi.org/10.14257/ijast.2017.109.06

Automatic Report Testing without Programming

Moukouop Nguena Ibrahim*[1], Tchinda Maxime Carlos[2] and Ndoundam Rene[3]

[1]National Advanced School of Engineering, University of Yaounde I – Cameroon
[2]Department of Computer Science, University of Yaounde I – Cameroon
[3]Department of Computer Science, University of Yaounde I – Cameroon
[1]imoukouo@gmail.com, [2]tmaximecarlos@gmail.com, [3]ndoundam@gmail.com

Abstract

Testing involves approximately 60% of the cost of software development, and includes the validation of reports, which plays an important role in decision-making in companies. Automatic report testing includes testing the content and the layout of the report. The conventional method used in testing reports leads to a development time of test cases being too high and impractical, and does not allow the automatic report testing of the content and the layout. In order to solve these problems and to simplify the testing of reports, we have developed three complementary approaches in this work. One ''visual'' approach, based on the visual comparison of some aggregations on the report. One semi-automatic approach, based on the use of aggregate functions to make an automatic validation of data. The third approach is automatic, derived from a mathematical definition of a test case introduced here and the use of checksum and metadata. This approach enables the fighting against reports regression and reduces the time of reports testing. As a proof of concept, we designed and built a framework al lowing the tester to do automatic testing of reports, without programming. Its implementation has permitted the automatic testing of reports, in a very economical way.

Keyword: *automatic report testing, aggregate, checksum, test cases, mathematical definition*

1. Introduction

The test has an ever increasing role in the development of software. To cope, research in computer science abounds and aims at facilitating the design of tests and improving their relevance. A study carried out by NIST (National Institute of Science and Technology) in the United States of America on the impact of inadequate testing infrastructure in software development has shown that this insufficiency cost about $60 billion to the U.S. economy in 2009 [32]. Regarding critical software, such as those that run the financial management systems of which one of the main functions is the production of reports, the consequences of a mistake can be very serious because there are human or natural disasters, or economic factors that are involved.

According to Solentive Software [22], when building a custom system, organisations need to consider that both the functionality and reporting are of equal importance. 20-40% of the budget should be allocated to reporting alone.

The particular case of automatic report testing remains an almost non-existing activity in the literature of software engineering. The main challenges concerning reports can be summarized in the following questions: how can we automatically validate data in a report? And how can we validate the whole report (data and layout) with automatic testing? How to be sure that there will be no regression of a report? Nowadays, reports are poorly tested,

Received (August 18, 2017), Review Result (October 11, 2017), Accepted (November 9, 2017)

the test process is time-consuming at a huge cost, and reports are usually subjected to regression, due to manual testing.

The most common traditional approach to report testing is the manual verification of the content and layout. This takes a lot of time and is not reliable. The time taken is high, we do not systematically test the reports after changes, and the regressions are numerous and costly. There is a more advanced approach which consists of automatically testing the content and to do a visual test of the layout. Automatic content testing poses difficulties related to the time taken to prepare the set of data to be used for the test and the determination of the expected results. A report of 1000 rows and 22 columns represents 22000 cells, which does not facilitate a cell-by-cell test. The visual test of the layout facilitates regressions and does not make it easy to see aspects related to the length of the text or variations in the number of columns. Although traditional testing approaches can be used for report testing, they do not give any answer to the practitioner on how to automatically test the content and the layout, what to choose to highlight the difficulties of formatting, and on how to automate the test programs. They just give a formal framework for testing any program, without bringing specific answers on which method to adopt to test reports. We are not only interested in finding errors in the reports, but also and especially in the time taken for the tests, and in the subsequent automatic detection of the regressions. The productivity of the tester preoccupies us as much as its efficiency.

In this work, we propose three complementary approaches. One "visual" approach, based on the display and comparison of aggregates on the report, which can be used in production. One semi-automatic based on the use of aggregate functions to make an automatic validation of data. These aggregates can be displayed on the report for visual validation or compared to expected values of the aggregates in memory to return an assertion. The third approach is automatic, based on the calculation of the checksum of the same report validated by the semi-automatic approach and comparing it to the checksum of the same report printed using the same input parameters but at a different time. We designed and built a framework allowing the tester to do automatic testing and other types of report testing presented here, without programming.

This article is organized as follows: a review of the literature on software testing in the general context, followed by a description of our different approaches to solving the issue on how to do an automatic test of a report. The third part focuses on the implementation of the approaches described above. Finally, we present the conclusions and perspectives related to this work.

1.1. Generalities on Software Testing

F. Xavier. Fornari, Glenford J. Myers and Mehdi Mirzaaghaei showed that software testing is a creative and critical activity that requires discipline and software engineering expertise. It is an activity where you have to imagine the scenarios capable of causing the software to fail, and to build the simulation environments, but it is wrongly perceived as the tester is usually at the end of the chain, which generally causes the delays [4] [5] [19].

Software testing would be the process of executing a program with the intention of detecting anomalies in order to validate its functioning [4]. In other words, it is done to validate the conformity with respect to the requirements (that is to say over the entire specification and design of the software) [4].

In general, constraints are tested to handle different classes of software. Hence the importance of respecting the basic test principles, namely: independence, paranoia, prediction, verification, robustness and completeness [4].

The particular case of report testing remains an almost non-existing activity in the literature of software engineering. On the other hand, the global economy depends on it because these reports play a decisive role for nations and companies. Many authors in their works address the idea of a test without however alluding to report testing. This is the case of F.Xavier. Fornari [4], Paul Ammann, Jeff Offutt [12] Glenford J. Myers [5] who presented with examples, the technical test bases and the quality criterion (Ease Readability, Reusability, Efficiency, Integrity) of software which in our case allowed us to test the implementation of our solution.

Other authors [2] [11] [15] [20] go further by speaking about automated software testing as a whole, while relying on concrete examples. This is the case of Lionel Duroyon [8] who presented environmental DTEST, a test Framework for distributed applications written in Python. It is also the case of Hernán Ponce de León, Stefan Haar and Delphine Longuet [20] who presented a theoretical test framework and an algorithm of test for concurrent systems specified with true-topic models generation. Some authors described the different variants of automatic testing and associated test cases; it is the case of Nisha Kaushal and Rupinder Kaur [11].

Some authors, such as Christopher Grandpierre [3], rely on design patterns in order to generate test data (see [9]), using UML or OCL diagrams to specify and generate test cases.

All this work does not address the specific challenges of report testing, as presented previously.

Based on the V-model, the different test levels that we can mention are: the unit test, the integration test, the conformity testing, the robustness test, the security test, the performance test and the regression test. Our work addresses the levels of conformity testing and regression testing, and can be used in some cases in security testing. Theoretically, the tests are prepared at the same time as the development in the corresponding level. Software testing can be grouped into two main groups (see [1] [4]) namely:

- The functional or dynamic test, also known as the black box test;
- The static or structural test, also known as the white box test.

In our case, we will look at the functional testing of reports.

1.1.1. Study of some Existing Software in the Field of Report Testing

As part of the study of existing tools to check for possible use in the reported testing. Several tools have received attention.

There are many softwares in this field, and we can mention among others, Jmeter, TestLink, Fit, Selenium, Mdal test module.

Jmeter is a software that allows you to test the scalability and performance of web applications.

In the case of report testing, it will serve to ensure that we can always continue to build and display reports when we have a large amount of data and several people connected. But it does not allow us to check the conformity of the displayed report without a complementary code for this checking, and how to efficiently build this complementary code is our main problem.

TestLink is a web platform written in PHP that allows you to organize your test cases as a test plan. In short, Testlink is a tool for managing test cases that can be associated with Selenium and JUnit in the case of the test of a report generated in HTML.

DataGen, TurboData, GS DataGenerator and Benerator are data generators for a database from the description of a table structure (they can create as many rows as desired). But these data cannot be considered as test data because we cannot control them and the generation of these data does not even meet the constraints of the specifications. In the best case, we can use these data in association with the actual test data for testing scalability in the case of our problem.

Selenium is a set of open source and free tools whose goal is to automate functional testing of web applications that can be associated with the main Test Framework such as JUnit [10]. Each of these elements is independently useless in report testing, but both associated may help for the testing of an HTML report.

MDAL Test module: it is a module of the MDAL (Megasoft Data Access Library) Framework dedicated to testing. It has many testing features, but no feature dedicated to report testing at the beginning of this work [17].

There exist other software [16] in the field of software testing that have no specific features for the problem of the testing of reports, such as: Fit, Junit, Sandboxie, the test environment for PayPal, SALOME, Jenny, RTMR, Schematron.

Limitations of Report Testing

Generalities on Reports

According to the business dictionary, a report is a document containing information organized in a narrative, graphic, or tabular form, prepared on ad hoc, periodic, recurring, regular, or as required basis. It is also defined as a document that presents information in an organized format for a specific audience and purpose.

It is increasingly a component of corporate balance sheets (Component of the annual report on sustainable development, for example) allowing to make a regular review of the company's strategy, the means put in place to serve this strategy and results obtained.

Report testing is the process of assessing accuracy and completeness of key reports:

- Accuracy is a precise representation of source documentation of the report (*i.e.*, items that appear on the report can be traced back to source documentation and confirmed)
- Completeness is the comprehensive collection of intended data on the report (*i.e.*, source documentation that should appear on the report, according to the reporting parameters, does, in fact, appear on the report).

The conventional method of report testing is to test as a unit, each of the variables and table cells constituting the report. This is also well illustrated in Java by Peter Fröhlich and Johannes Link [13]. This method has the disadvantage that it takes a very long time to be built, and it is not useful to detect some errors due to anomalies or inconsistencies on data, in the production phase.

The Standard Method of report testing consists of:

- Execution of the report with the given period or parameter;
- Running the data selection query over the same period on the database;
- Comparison cell to cell of report with the result obtained from the query;
- Visual validation of the report display.

This standard method has two variants:

- Manual validation of cell by cell of the data and visual validation of the display
- Automatic validation of cell by cell of the data and visual validation of the display

During the report testing, one needs to fight against the regression in reports. Regression comes from future changes in the report's code or in other components or systems used by the report (frameworks, DBMS, common components...).

2. Proposed Methods

Description of the Approach

For the report testing, we define the following three approaches:

- The first one based on the use of aggregate functions to manually validate the data without validating the display.
- The second one based on the use of aggregate functions to automatically validate the data without validating the display.
- And the third using the validated results of the first to perform an automatic test on the calculation of the checksum of the report.

In our approaches to report testing, a report is seen as an object to which we can link variables and datasets (sets of data stored in memory) that can be referenced externally by the tester to perform automatic tests on data. Our approach is to separate the variables and datasets. For datasets, we use the aggregate functions and for variables, we use some comparison parameters.

Aggregate for Visual Control

The approach presented in this paper constitutes a means to test "manually" data reliability in a cheap way, even in production. This technique is based on two pillars:

- Define two variables which represent data aggregation (aggregate) on different datasets, according to a well-defined criteria, and which must have the same value for the data used by the tested report;
- Display these variables and their difference on the report, in other for the end user to check the report by seeing that the two variables have the same value.

An aggregate is defined as the value of an aggregation, is an accumulation of data for purposes such as statistical analysis. Aggregates in the case of report testing are generally statistical functions namely the minimum, maximum, sum, average, standard deviation, etc.

As an illustration of this approach, we can use the following example. You have two tables: account (id, designation, debit, credit) and movement (id, account_id, movement_date, debit, credit). You also have a constraint stating that the sum of credit for all movements of an account is equal to the credit of this account, and a similar rule is valid for the field debit. On a report displaying all the movements of an account, one can display a variable representing the sum of credit of the movements of this account, near the value of the field credit of the same account. If there is a difference between the two values, one can conclude that the report or the data is wrong. One can say that, having the same value does not prove that the report is correct. That is true. Normally, the purpose of a test case is not to prove that the software is correct, but to find a fault. We will address the issue of the reliability of this approach in the sequel. Now, one can note that with a good choice of the aggregate, it is a very nice feature for a report to display some variables for a quick visual validation, and it helps, even in production, to detect some problems on data or on results, or even some frauds. The main interest of this approach is to allow end users, to easily detect errors in production.

Architecture of Report Generator

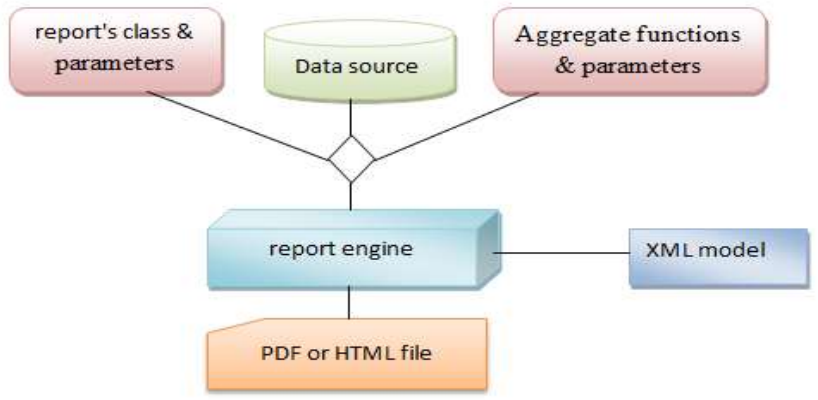

Figure 1. Architecture of a Report Generator with Aggregate Calculations for Visual Validation

Figure 1 shows that once the data collected following the parameters of a report's class and associated with a model, aggregate functions are responsible for calculating all the aggregates according to their parameters, before submission to the report's engine, who will display these values on the report.

In this approach, the validation of the report can be made by an end-user thanks to the fact that the report is developed in such a way that any person without necessarily being a tester can validate it.

Semi-automatic validation of the report

This method only allows data validation, but does not validate the display. The validation of the report's data is done automatically by the calculation of aggregate functions, whose values will be compared to those of the tester who is responsible for providing the test data. The display of the report is validated by a user. At this point, we use automatic validation for data and user validation (visual) for report's layout.

This approach can be defined by the architecture of report generator below.

Architecture of Report Generator

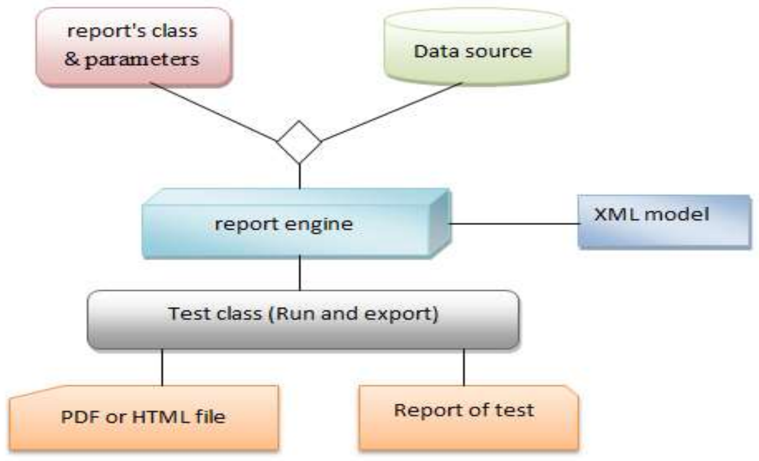

Figure 2. Architecture of a Report Generator with some Automatic Testing

Figure 2 shows that once the data is collected following the parameters of the report class and associated to a model, the test function will calculate all the aggregates in the test cases and compare them to those provided by the tester in test data. Finally, the tests report is sent along with the printout of the report. This approach allows automatic validation of the report's data using aggregates. The aggregates can be computed on all the data in the report or on subsets.

“The choice of using filters and aggregates such that the sum allows leaving the subset respecting the aggregate.”

To cheat or to fault the test, it would be necessary to modify the data while maintaining the aggregates. It involves the resolution of the SUBSET-SUM problem. This problem of the sum of subsets is NP-complete; therefore it is considered difficult to solve algorithmically. It can be seen as a special case of the knapsack problem [24].

What about the column with non-numeric data? It remains possible to calculate some aggregates on these columns (min, max, count of a value), and to use filters to build subsets. It is also possible to build a concatenation of all the values of the column, and to compute a checksum of this concatenation as the aggregate. Modifying a value or making a permutation will probably lead to a different checksum. Let us recall that the purpose of the test is not to know which particular unit is incorrect, but whether the report is good or not. When it is established that the report is not good, a dichotomous approach of calculating aggregates on the subsets permits that from the detection of an error it becomes easy to find the cell that has a problem.

One can note that it is also possible to use this approach (with checksum) with numerical data, which represents the main critical data type in financial reports.

Automatic Test: Using the checksum of the report

The approach here is to have a way to calculate the checksum of a report after its semi-automatic validation made above (automatic data validation and visual display) and compare it with the new checksum of the same report using the same parameter values each time it is printed. This checksum calculation permits to move towards an automatic test which rather becomes an instrument to fight against report regression.

The checksum is used to combat the regression in terms of data validation and display validation. Because any change in Visual terms or content will alter directly the value of the checksum. In the case of our implementation, we chose the CRC32 algorithm which permits to return a 128 bit checksum for a file passed as a parameter.

If there are variations of contents or shape in a new version of the report, the checksum will be recalculated for this new version. We can then decide to keep only the last version of the report with its checksum, or to keep the codes of various versions, with the test data suited for every version, as well as the corresponding checksum. Such modifications can be common when the software is in development, but are very rare for the stable software.

Description of the Approach

The test scenario has two main steps:

1) After a semi-automatic validation of the report, compute the checksum of the report, excluding some undesirable metadata

2) Build a test case for automatic testing, using the computed checksum as a value to be compared to the checksum of the built report at any time.

The test scenario is defined as following:

- build the report as a file (pdf, html...) on a given period or with specific parameters;

- Calculate the checksum of the report;
- Compare the invariant with that of the report validated by the semi-automatic test.

This method still poses a problem related to the printing date of the report. For example, if we have a report that has a printing date, when it will be built on another day, the printing date will change. Therefore, the checksum will change, and the test will fail.

Solution found: Set a time provider for the test engine such that when a report is started, instead of using the system date, it uses the date provided by the time provider. This time provider must be configurable and we should make sure that every time the report is launched, the time provider is initialized to the same time.

For our problem, we use the variable timer.start which contains the report printing date. This variable can be passed in the parameters of the test case for the tested report. This implementation has been set up to test the report validated by the first approach and to fight against regression. This approach can be defined by the report generator architecture below.

Architecture of the Test Generator

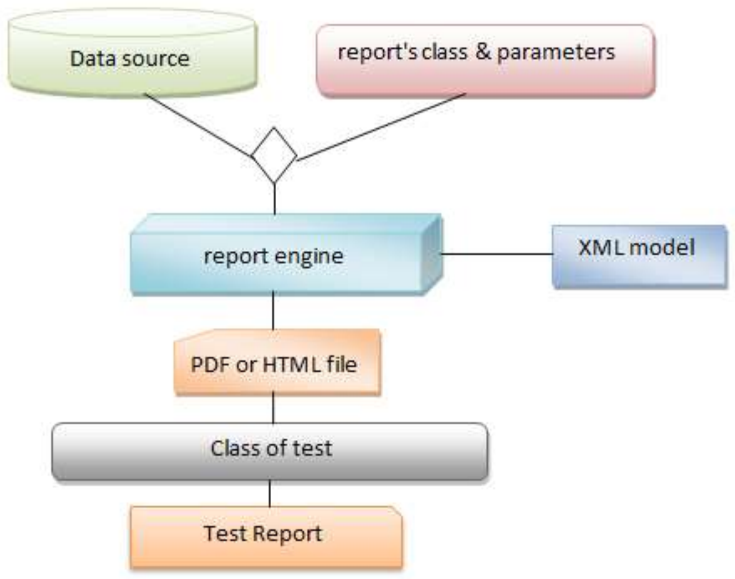

Figure 3. Architecture of the Report Generator Engine with Checksum Calculation

Figure 3 shows that once the data is being collected following the parameters of the report class and associated with a model, the report is generated. The result obtained is then passed as input to the test function that calculates the checksum and compares it with that of the previously validated report which is passed as a parameter to the measure of a test. Finally, the tests result is returned.

For reports in HTML format, there is no undesirable metadata. For PDF reports, one has to deal with some metadata as undesirable metadata. For PDF documents, It is indeed six free text fields that describe the document and its means of production (Title, Author, Subject, Keywords, PDF Producer, Application) and two fields of date type (creation date, modification date) [21]. The first six fields may be optional, while the last two are required. Thus, in the context of calculating the checksum these will fail a test case even for two reports that are printed with a difference of a second in the creation or modification dates. The purpose here is to compute the checksum of the report without taking into account this metadata.

As for reports produced by itext2.1, we notice that this metadata are generally placed at the end of the PDF file, after the report's data and layout. Therefore, we can compute the checksum of the part of the file before these metadata. In order to do it, we just have to find the beginning of these metadata, by looking for a unique string, called the metadata delimiter, which is always before these metadata (the producer as an example). In order to fight against the modification of the metadata delimiter following the change of the producer of the report in the future, we will pass it as a parameter of the test case.

Using a time provider with a start time parameter, and calculating checksum without undesirable metadata, we can completely automate the report validation after a semi-automatic validation. We stated that semi-automatic validation relies on the use of aggregates in order to verify the correctness of the data displayed on the report. It is obvious that having the same aggregate value than the correct data does not mean that the result is correct. Therefore, one can have two questions: why do we use aggregates instead of a whole dataset comparison? Can we rely on these aggregates?

For the first question, one can note that there are many reports including tables with hundreds of rows, and more than six columns. For such a report, one can have more than one thousand values in a table. Trying to validate these values one by one will be very costly in time and money. It is also very difficult and can lead to many errors. We have to find an economic way to check the validity of reports.

For the second question, the answer depends on the choice of the aggregates. With a good choice, one can rely on them to have a good checking of a report. Let us consider one column in a table, with numeric data. If we use as aggregates for this column the sum of its values, the sum of the squares, the standard deviation and the sum of the cubes, it is reasonable to think that by using another dataset we can have the same values for these aggregates, these datasets have with a high probability the same values for all the rows of this column, even in different order (permutations).

If we want to be sure that the same results are not due to a permutation, we can use at least two methods. The first method is to compute the same aggregates on some subsets of the dataset, defined by some filters. If a permutation does not maintain all these subsets globally invariant, it will lead to different results. For the second method, one has to identify each row of the dataset by a unique number. The aggregates are computed using the product of the row number by the value as elements to aggregate. Using the same aggregates, any permutation of column values, will lead to different results with a very high probability.

3. Results

These three methods have been implemented in various projects such as NEXUS++ (Tracking System for the transit of goods on the Cameroon territory), SACOTAB (Automatic follow-up of the contracts of objectives and dashboards), SIBERIA (business management system of a Real Estate Company in Cameroon) and many others. The nexus++ project contained more than 254 reports, more than 100 of which were very complex. For this project, the aggregate approach quickly became the only economically sustainable and efficient way to test the reports. The SIBERIA project had several critical financial statements, and aggregate techniques were used to validate the accuracy of the reports in test and production. Better still, visual display of aggregates for comparison has been beneficial in this project, highlighting anomalies whenever attempted fraud or bugs in the treatments resulted in anomalies in the data. By making the reports so reliable to bring out the anomalies that went unnoticed before, the revenue improved by about 50%!.

4. Discussion

Statistical Analysis of Test Cases Performed

For the purpose of this article, the implementation of each of the following approaches was experimented on (05) reports depending on the type of the report. This yielded the following statistical results:

Approach	Type of report generated	Number of test cases	Number of successful test cases	Number of failed test cases
Use of aggregates for manual validation of data	HTML	16	16	0
	PDF	21	19	2
Use of aggregates for automatic validation of data	HTML	75	71	4
	PDF	87	85	2
Use of checksum	HTML	9	9	0
	PDF	8	8	0

Analysis: as concerns the reported testing by the aggregate approach, we achieved a 95,98% result for HTML and PDF reports. Regarding the reported testing using the checksum based approach, we have achieved 100% results for HTML reports and thanks to the elimination of certain metadata, a score of 100 % for PDF reports.

Let us recall that these approaches were successfully used in projects involving more than 250 reports, which permitted us to realize how painful and unproductive it was to work with the standard approach.

Comparison between the conventional method and the proposed methods

The conventional implementation in an automated environment demands great effort in the production of test data, due to the fact that it is necessary to produce the expected results cell by cell. So for one that produces the test data, it becomes very laborious. We experienced that, when we wanted to do the test with the conventional method, we had to go cell by cell to compare expected results and actual results.

Either a report to be tested. For the cases of an automatic or semi-automatic test, we used a generator of data test on which we simply applied our aggregates. Let T, the time of generation of a datum of the test of a cell in an automatic way. Let T1, the average time to machine calculation of one aggregate. Let T2, the average time to manually identify a good aggregate. Let T3, the time of generation of a datum of test of a cell in a manual way. Let T4, the run time of a manual comparison between a datum obtained from a cell and the expected datum from the cell. Let T5, the run time of an automatic comparison between a datum obtained from a cell and the expected datum from the cell. Let p, the minimal number of the aggregates to be calculated. Let T6, an average travel time of one cell of the report for the calculation of the checksum.

The complexity of the production of the test data has two main components: the complexity of the production of the Input and the complexity of the production of the output. Let n1 the number of report data to be tested cell by cell (master fields in a master-detail report for example). Let N the number of cells in the tables of the report (these cells can be tested using aggregates). Ia the complexity of the automatic generation of input data, Im the complexity of the manual production of input data, Cs the time to compute the checksum of a report, P2 the time to identify aggregates to use to compute differences, THV the time for a human validation of the display

On tabular reports of many pagses, N can be easily greater than 10 000 (500 rows * 20 columns), while p is rarely greater than 2*columns.

Let's use the following notations for different methods:

Standard (S): Manual validation of cell by cell of the data and visual validation of the display

Detailed (D): Automatic validation of cell by cell of the data and visual validation of the display

Visual difference (VD): display the differences between aggregates supposed to be equal (visual validation of the report by a user)

Semi-automatic (SA): using aggregate functions (automatic validation of the data and visual validation of the display)

Semi-automatic with differences between aggregates (SADA): using aggregate functions, automatic validation of the data and visual validation of the display + display of differences between aggregates

Automatic (A): use of the report checksum

Automatic with differences between aggregates (ADA): use of the report checksum and display of differences between aggregates.

Method/Feature	S	D	VD	SA	SADA	A	ADA
Human complexity of the test run	THV+ (N+n1) *T4	THV	THV+ P*T4	THV	0	0	0
Enables to develop the test without programming skills	Yes	No	Yes	Yes	Yes	Yes	Yes
Enables to test the regression of the data	Yes, But impossible in practice on a project with dozens of reports, because too expensive in time and money	Yes	Yes	Yes	Yes	Yes	Yes
Enables to test the regression of the display	Yes, But impossible in practice on a project with dozens of reports, because too expensive in time and money	No	Yes	No	No	Yes	Yes
Validates the consistency of the report data in production	No	No	Yes	No	Yes	No	Yes

complexity of production of the test data	Ia+ (N+n1) *T3*	Ia+ (N+n1) *T3	Ia+ n1*T3	Ia+ n1*T3+ p*(T1+T2	Ia+ n1*T3+ p*(T1+T2) +p2	Ia+ n1*T3+ p*(T1+T2) +Cs	Ia+ n1*T3+ p*(T1+T2) +Cs+p2

One can note that using aggregates in report testing is very useful. It leads to a great reduction of human workload during the construction and the execution of tests cases, while allowing the regression testing of a report. Let us note that the use of aggregates in finance for the validation of reports is a major element that is found at least in the verification of the balance in accounting (sum of credits - sums of debits = 0).

The standard method does not enable us to validate the consistency in production, Except for gross errors. We were able to verify it on hundreds of reports. It is important to include on the report elements of validation allowing a non-expert to realize problems.

Although the production of the tests data is "manual", it is done only once, and allows the repeating of the test automatically as many times as necessary. The main gain here lies in the execution of the test, and in the automated detection of regressions. In addition, using the aggregates to verify accuracy reduces the time required to complete the test program.

Limitations of the Proposed Methods

Despite the reduction of work done by these methods in the preparation of the tests, the generation of the test data consistent with the treatments supposed to produce them remains a Long and difficult task to test reports. Reflections must be made on how to better automate this generation. When the calculation of the aggregates uses the powers (the square or the cube of the displayed values) and the number of lines is high, the aggregate can rapidly exceed the amplitude of the type of long integer data when integers are used. It may then be necessary to resort to numerical types with arbitrary prediction. The proposed techniques are not intended to immediately give the wrong cell for a report. In order to have this information, it is necessary to apply them in the context of a dichotomous search, reducing by half the total number of lines. Evolutions should be considered to automate this research once it has been established that the overall report is bad.

5. Conclusion and Perspectives

At the end of our analysis, it appears that the problem which we faced namely automatic report testing is the execution of a report to ensure that it displays the correct results in the expected form. The standard approach known to test reports does not enable us to automatically test the display, and to validate reports in production. In the first part of this article, after defining the context and the problem we face, we presented our motivations and various stake of the solutions which we have proposed. Then followed the different implementations of the distinct approaches available in the MDAL test Framework engine.

Our goal was to provide methods for automatic report testing. We have provided three complementary methods, and extended them to five methods. The first method allows a quick visual validation in a production environment. The second method is a semi-automatic test method that allows automatic testing of the content of the report and not the display in a cheaper way, using aggregates and filters. This method allows an automatic validation of the content and a visual validation of the layout by the user. The third method is dependent on the second one which after a validation of the data and the display of the report can be considered as a perfect report and allows fighting against regression. It allows an automatic validation of the content and the layout of the report, using the result of the second method. This method lies on the use of a time provider instead of using system time provider, and on the use of a part of the generated report, avoiding the use of some undesirable metadata. We have also designed and built a framework for report testing, which allows software testers to describe test cases and validations checking without writing a new code. The extensions of these methods allow the validation of the report in production.

All methods were successfully tested with HTML and PDF reports. But they do not permit us to test a report since the production of test data is displayed automatically. Besides, the display is the part in which we found no validation method that is automatic. As perspective, we will now have to think about a way to automate the production of real test data that would validate a report from the point of view of the content. This will help reduce the time that testers take to produce real test data. Moreover, we will have to compare the different set of aggregates in their ability to detect errors on reports. We will also have to deal with reports which are directly printed on a printer or drawn on a canvas. We must find a way to automatically validate the display of report.

Acknowledgments

We thank the Government of Cameroon for its support during our research.

We thank MEGASOFT SARL Company for allowing us to undertake this work.

We thank CETIC (African Center of Excellence for Information Technology and Communication of Yaounde, Cameroon) for its expertise.

References

[1] B. Meyer, I. Ciupa, A. Leitner and L. (Ling) Liu, “Automatic testing of object-oriented software, Chair of Software Engineering”, ETH Zurich, Switzerland, International Journal of Computer Science & Engineering Survey (IJCSES), vol. 2, no. 1, **(2011)** February.

[2] C. Binnig (SAP AG), D. Kossmann (ETH Zürich) and E. Lo (The Hong Kong Polytechnic University), “Towards Automatic Test Database Generation”, IEEE Data Eng. Bull, **(2008)** January.

[3] C. Grandpierre, “Generation Strategies Automatic testing of a behavioral mode UML / OCL”, These doctorates at U.F.R. Science and technology The University of Franche-Comte, **(2008)** July 17.

[4] F. Xavier. Fornari., “Introduction to Software Testing”, Esterel Technologies, **(2011)**.

[5] G. J. Myers, C. Sandler and T. Badgett, “The Art of Software Testing”, Third Edition, 240 pages. Wiley, **(2011)**.

[6] G. Tassey, “The Economic Impacts of Inadequate Infrastructure for Software Testing Final Report”, 309 pages, Wiley, **(2002)** May.

[7] H. Tahbildar and B. Kalita, “Automated software test data generation: the direction of research”, International Journal of Computer Science & Engineering Survey (IJCSES) Vol.2, No.1, Feb 2011

[8] L. Duroyon, “DTest: A Framework of Tests for Applications Distribuées”, Onera Center of Toulouse, **(2008)** June.

[9] L. Cardenas, “An automatic test generation from requirements specification”, University Of Quebec In Outaouais, **(2007)** August.

[10] M. Nebut, “Introduction à Junit”, University of Lille, **(2007)** September.

[11] N. Kaushal and R. Kaur, “Comparative Analysis of Various Automated Test Tools for Flex Application”, National Institute of Technical Teachers' Training & Research, Punjab University, Chandigarh, Panjab, **(2009)**.

[12] P. Ammann and J. Offutt, “Introduction to software testing”, Cambridge University Press, **(2008)** February.

[13] P. Fröhlich and J. Link, “Unit Testing in Java”, Elsevier Inc., 2002.

[14] P. Ranjan Srivastava and K. Baby, “Automated Software Testing Using Metaheuristic Technique Based on an Ant Colony”, Automated Software Engineering, **(2011)** June 14.

[15] R. A. Demillo, Purdue. University and A. Jefferson, UTT Clemson University, “Experimental Results from an Automatic Test Case Generator”, ACM Transactions on Software Engineering and Methodology Homepage archive, vol. 2, no. 2, **(1993)** April, pp. 109-127.

[16] W. Piez and D. Lapeyre, “Introduction to Schematron”, Mulberry Technologies, SmartBear Software Inc., **(2008)** November.

[17] M. Ibrahim, “Guide of the MDAL developer V4.3”, **(2012)** November, pp. 318-336.

[18] M. Zhivich, “The Real Cost of Software errors”, MIT Open Access Articles, **(2009)**.

[19] M. Mirzaaghaei, F. Pastore and M. Pezzè, “Automatic test case evolution”, Software Testing, Verification and Reliability, **(2014)** April.

[20] H. Ponce de León, S. Haar and D. Longuet, “Model-based testing for concurrent systems with labeled event structures”, Software Testing, Verification, and Reliability, **(2014)**.

[21] F. Bouquet, P.-C. Bué, J. Julliand and P.-A. Masson, "Generating tests based on dynamic criteria and by abstraction", LIFC, University ofFranche-Comte, 16 Route de Gray F-25030 Besanccon Cedex France, {bouquet, bue, julliand, masson}@lifc.univ-fcomte.fr4, **(2011)** February.
[22] Solentive Software, "The importance of reporting: why it demands 20-40% of your budget", whitepaper, **(2012)** March 13.
[23] H. Wang and C. Zaniolo, "User-Defined Aggregates in Database Languages", Springer, **(2001)** April 13.
[24] M. Sipser, "Introduction to the Theory of Computation", 2nd Edition, Course Technology, **(2006)** February 15, pp. 295-298.