Volume 3, Issue 8, August 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

An Approach to Develop Validation Framework for Electronic Invoice

Sucheta Thakur
Department of Information Technology
Pune University, Pune, India

Dr. Emmanuel M
HOD, Department of Information Technology
Pune University, Pune, India

Abstract- Software keeps on evolving over time due to certain enhancements done in business rules, which often results in unexpected modification of data. Regression testing has become a vital activity to record adverse effects of changes done in latest build of software. Since Organizations adopt business rules of varying complexity it becomes a necessity to discover unexpected consequences caused after modifying those rules. This work proposes a method to device a solution for capturing variations in database caused due to modifications in existing system. Validating an invoice has become an important activity during invoice processing. Currently; there is lack of testing strategies for processing invoice rules in telecom domain. This method evaluates the database against the altered rules in invoice processing. The approach was validated through implementation of a testing framework on invoice rule sets of telecom dataset. Two different versions are compared and test cases for newer version are generated from older version. We have successfully tested our framework and observed the results.

Keywords — Regression Testing, Test Case Selection, JUnit, XML, EDI invoice.

I. INTRODUCTION

Regression testing is usually done during maintenance as it is necessary when a program has been changed. If adaptive or perfective maintenance [7] were performed then program specifications were modified and if corrective maintenance was performed then the specifications may not have been modified. Regression testing verifies changes and checks other parts of the software that are affected. During the coding stage of software development lifecycle, testing becomes important. Many common coding errors occur when business rules is formed or modified. Often, rules are required to change to resolve customer's issue. Resolving of tickets causes frequent changes in invoice reader modules.

Various components in system increase the complexity of database applications. Database applications play an important role in today's commercial systems [5]. Most of these applications get modified constantly which results in its increased complexity resulting in frequent changes of configuration. Testing database applications is extremely important since inputs are processed according to rules. Besides the modified version of rules may not only produce modification in unexpected values, but also cause miscalculation. Incorrect data are dangerous since they may be an input of other processes of an application, which may cause malfunction.

Most database applications are subject to constant change; for instance, business processes are re-engineered, authorization rules are changed, components are replaced by other more powerful components or optimizations are added in order to achieve better performance for a growing number of users and data [5]. It has become quite expensive to carry out tests for new version of software release.

In telecom, there are still certain errors while processing bills. One of the factors is the size and structure of invoices which is very complex. Other reasons that cause errors in invoice processing can be multiple charges for same call. Service charges based on volume might not be applied correctly. Some charges may vary from city to city, so taxes may be applied incorrectly. According to a survey [8], telecommunications expenses rank as one of the five expenses for most companies. Though it has evolved from decades, yet it lacks data validation. This paper presents a practical approach for regression testing of complex database applications. Such applications are important to many organizations, but are often difficult to change and consequently prone to regression faults during maintenance. They also tend to be built without particular considerations for testability and can be hard to control and observe. A practical solution for functional regression testing is proposed that captures the changes in database state (due to data manipulations) during the execution of a system under test. The differences in changed database states, between consecutive executions of the system under test, on different system versions, can help identify potential regression faults.

This paper is organized as follows. Section 1 explains the need testing for analyzing regression. Section 2 reviews Literature Survey. Section 3 discusses our approach in detail. Section 4 summarizes the results of case study. Section 5 discusses implementation issues and section 6 concludes the approach.

II. LITERATURE SURVEY

A lot of work has been done on regression testing [1, 2, 3] were examined and reviewed. An overview of all regression test selection techniques have been researched from all previous papers in [2]. Regression testing using VDM ++ [3] specifications have been explained with an objective to reuse and reduce test cases. Authors have very well classified the test cases from original test suite into obsolete, re-testable, and reusable test cases and created a new test

suite for new version containing meaningful test cases. Test cases not required in new version have been successfully eliminated from test suite. Comparison of the techniques has been classified and compared. A unit test approach has been proposed [1] for validating database schema. Database modifications support database refactoring as well as requirement modifications. Comparing to our approach, it is a difficult task to change large datasets. Instead, it is better to validate a dataset by persisting invoice into database and perform unit testing.

III. DESIGN AND IMPLEMENTATION

Fig. 3.1 represents a model of testing framework with deterministic finite automata. There are eight states through which our validation framework passes when it runs in JUnit framework. The states are defined as:
A: Invoice Reader alpha version; B: Invoice persisted in database; C: Parse Expected Syntax XML; D: Syntax comparison; E: Parse Expected Semantic XML; F: Semantic Comparison; G: Test Cases Passed; H: Test Cases Failure; I: Dead State. The grammar for the following framework is 111(0.1)*1(0.1)*(1+0).When framework runs test selector module it returns classified test cases. This can be elaborated with set theory.

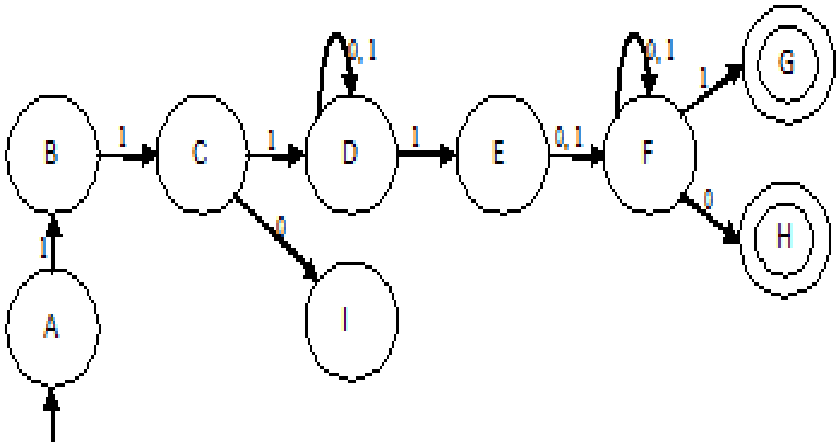

Fig 3.1 Finite Automata of proposed Framework

Definition: Let I be finite set of invoice inputs. Set I = {lf1,lf2.....lfn} where 'I' is an input set consisting of 'n' number of invoice load files in test suite . Let 'Evg' be a set of expected values that can be derived using function δ (e) i.e. generating expected value and 'Avg' be a set of actual values derived using β (q) i.e. generating actual value .Let γ be a comparator function. An output of the framework is 'o', where $o = \{T_{\text{Retestable}}, T_{\text{Reusable}}, T_{\text{Obsolete}}, V_r\}$ .TRetestable is a list containing those attributes whose value differs. 'TReusable contains those attributes whose value remains constant and TObsolete contains those attributes which are no longer available in newer version and hence, are termed as obsolete [1]. "Vr" is defined as

$$\text{Vr} = \gamma\,(\text{Avg}, \text{Evg}) = \begin{cases} true & if(Avg = Evg), \\ false & otherwise \end{cases}$$

Invoice files consist of different formats as demanded by customer. Each format has different reader modules for reading an invoice. For e.g. an EDI invoices will be read differently as compared to csv format. 'Evg' represents a set of expected values that have been stored in an XML file. An example of EDI invoice is considered as:

Since an EDI is divided into header, summary and detail, lf1= {H1, S1, D1}; Header par H1= {h11,… h1l} consists of 'l' number of header fields. Summary part S1= {s11, s12……. s1m} consists of 'm' number of summary fields. Detail part D1= {d11, d12, d13……..d1n} where D1 consists of 'n' number of detail fields. The validation can be represented in logical terms as:

$I_j = T \ni H_j \wedge S_j \wedge D_j = T \forall h_{1x}, s_{1y}, d_{1z} \in (H_j, S_j, D_j)$ Where $1 \leq j \leq n$ and 'n' is number of invoice formats considered in test suite.

XML file is maintained by developer where expected values of invoices are stored. An EDI invoice contains large number of transactions. Some values which are very critical are considered in XML initially, so that any modification can be recognized in early stage of development. Various XML files are created for different invoices belonging to different vendors, since for each vendor different rule sets are defined. The XML file we have used can be illustrated as below in fig 3.2.

XML file shown contains scenario name along with SQL queries and its expected results. In case some mismatch occurs, the failure attribute is returned to the developer. Figure 3.2 explains the structure of XML file. An example taken for consideration is an EDI file. Let us consider a sample rule set which represents the invoice number to be picked up from segment A (suppose) but then vendor decides to pick up invoice number from segment B. This causes changes in rule sets. If this rule set is used to manipulate another variable (say V), rules for V also needs to be changed. Rule sets vary from being as simple as a statement to being the addition of many segments to produce a result. Since a change of rule sets requires human logic, a unit test framework built for validating rules will help finding out regression as well.


```
<?xml version="1.0" encoding="UTF-8" ?>
<Scenarios>
  <Scenario name="Invoice Header">
   <Header >
          Select COLUMN_A, COLUMN_B from
          invoice_header    WHERE id = {ID}
   </Header >
   <DisplayFailureMessage hasParameter="true">
          The Header Rows Scenarioion is Failed for
          FIELD_NAME of COLUMN_A {0}:
   </DisplayFailureMessage>
   <ExpectedValue>
          <Rows>
            <Row>
                  <DATA>Value Of
Column_A</DATA>

                  <DATA>Value Of Column_B</DATA>

            </Row>
          </Rows>
   </ExpectedValue>
  </Scenario>
```

Fig 3.2 Structure of XML file

3.1. Invoice reader Alpha Version and Beta Version

We compare alpha version and beta version of invoice reader in our proposed approach by checking the specification of invoice. These versions contain Rule sets to read invoice. Rule sets are subject to change in case vendor changes the format or some up gradations are made. Our validation framework will compare the specification syntactically and semantically. Invoice will be read through invoice reader alpha version and stored in database. Validation framework will get details of invoice from database as actual values. First validation is syntax validation which checks for presence of elements, segments and other segments which depends on it. Change in syntax validation leads to semantic validations as well. For semantic comparison equivalent values are validated.

3.2. Syntactical and semantic validation

In telecommunication industry, it is necessary to determine affects caused due to change in rule sets. Comparing an invoice format syntactically and semantically not only gives confidence, but also helps in detecting change in an earlier stage of invoice processing.

3.3. Syntactical validation

Initially, we compare two versions of invoice readers and find change in the syntax. Syntax comparison involves checking the presence of segments. If there are changes in syntax found, semantic test cases for that segment are added for validation (since we are making semantic validations for those calculations only which are prone to error).

3.4. Semantic validation

Syntax comparison checks only for presence of segments, so semantic validation is more important as it involves manipulation of values.

3.5. Expected Value

These are values stored in XML file. Values obtained by running "InvoiceReaderAlphaVersion" as shown in fig 3.3 are stored in XML file with an assumption that those values are correct.

3.6. Actual Value

Referring Figure 3.3, implementing an invoice on "InvoiceReaderBetaVersion" gives actual values, since "InvoiceReaderBetaVersion" is updated version.

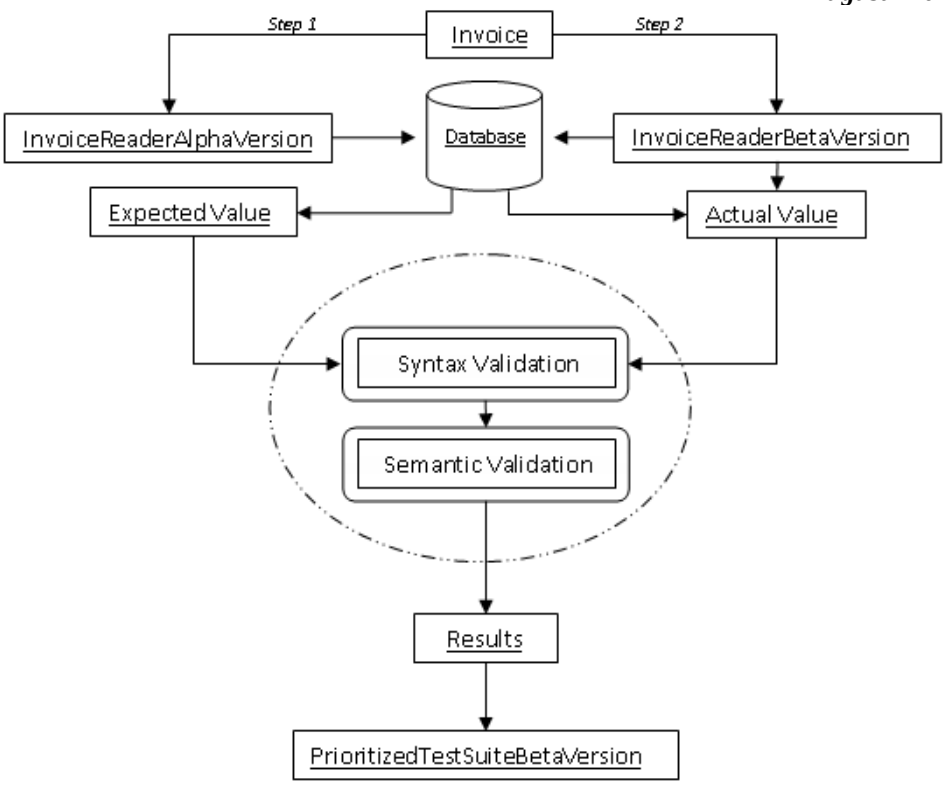

Fig 3.3 Implementation model of Invoice Validation Framework

3.7. Algorithm of proposed approach

In our proposed approach each invoice is identified by unique invoice id in database, which is system generated. The original rule sets version through which invoice is initially loaded is termed as Alpha version and after change, the newly defined rule sets is termed as Beta version.

When validation starts, invoice is parsed through rule sets and persisted into database. Syntax of segments are extracted and stored in xml file. For semantic validation expected values are stored in a XML file and compared against the values stored in database loaded through new version of rule sets. Comparator list passed test cases in a list of unchanged test case and failed test cases in a list of changed test case. After validation is performed, a report is generated which gives brief summary of test cases executed from our proposed framework and also returns a list of test cases which resulted in changed values as well as unchanged values. Developer can view output in an excel report. List of test case with changed values are given higher priority than the list of those test cases with unchanged values. Developer, either works on rules, or upgrade XML file to newer version depending on the kind of regression found.

Results: After implementing test suite, an excel report is generated which displays the values along with scenarios, making analysis easier. Invoice from the database gets deleted after validations are performed by using SQL queries.

3.8. Implementation of our Proposed Approach

The proposed framework for validation is integrated with existing system. Test cases are stored in an xml file maintained along with queries and expected value. This is an expensive process since it involves the knowledge of critical manipulations to be considered for validation. After generating test cases for alpha version of invoice reader we compare the results of alpha version specification and beta version specification. After analyzing these changes, appropriate test cases from alpha version test suite are selected for beta version.

3.9. Types of Possible changes in format

Some of the expected changes that may lead to change in regression test case for beta version are as follows:

3.9.1. Change in syntax

Changes in syntax of invoice format result in addition or deletion of elements or segments. Modification and implementation of new syntax rules involves in re-testing the test cases.

3.9.2. Changes in semantics

Since our approach has been limited to telecommunications industry we are checking the semantics of invoices from the database according to newly implemented rule sets. In case a new rule set involves concatenation of two segment values instead of picking a single segment value previously, there are changes in operations. Changes in operations can be detected through change in values.

Addition of rule sets occur when new rules need to be implemented, which was not present in alpha version previously. This leads to addition of test cases for beta version of invoice reader module as well.

Deletion will occur when a particular functionality is no longer required for beta version. In this case, the test cases related to deleted functionality is removed from the test suite.

3.10. Test Case Format

The alpha test suite should consist of the test cases for validating rule sets and a rule set can have multiple test cases because the dependency among segments can affect some unexpected values also. Sample Test Case for alpha version has been shown in Table 1.

3.10.1. Selecting test cases

Test selector module selects test cases for beta version based on validation result. Initially, test suite of alpha version will be the test suite for beta version. But after comparing the results, regression test cases for beta version will be updated. Test cases selector module will select test cases for beta version.

Table 1 Test case for alpha version

Test Case ID	Attribute	Input Segments	Expected Output
1	Invoice Date	BIG01	30022012
2	Invoice Number	BIG02	76445
3	Communication Number	PER04	78659467590
4	Communication Number Qualifier	PER03	TE
5	Quantity	QTY02	786
6	Identification Code Qualifier	NM108	41
7	Current balance	TDS+BAL-P-PB+BAL-A-NA+BAL-M-TP	76859

3.10.2. Reusable test cases

Test cases which produce same result for alpha as well as beta version are termed as reusable test cases [1]. These test cases are valid for both version .If there is no change in syntax rules and semantic rules these test cases are constant for alpha and beta version.

3.10.3. Obsolete test cases

After generating test cases for beta version, the generated test cases are classified by comparing two versions of invoice readers. There is some test cases found that was valid for alpha version and no longer valid for beta version. For example there is a segment whose syntax in alpha version is different from syntax in beta version or the semantics of the segment has changed and new segment is specified for beta version then the test cases defined in alpha version is obsolete and it is of no use for beta version.

3.10.4. *Re testable test cases*

Some test cases are defined to test the evolved part of invoice reader version. These test cases are re testable [1] test cases to test the modified part of invoice reader. For example if a semantic of segment changes, the segment does not produce expected results. Hence, test cases written are useful to test the beta version of invoice reader.

IV. CASE STUDY

In this section, we present a case study as mentioned in Table 1 to verify our proposed technique. Table 1 contains some sample test cases. Invoice Reader is basically a class for invoice rules sets. Beta version of our specification contains some changes like addition of elements, deletion and modification of segments etc. In this example for regression testing we have generated test cases for the alpha version and then selected test cases for the beta version. We have the alpha specification as input. On the basis of comparator module we find out test cases for the beta version and apply our approach on validation framework. We compare actual and expected values of both versions and generate regression test case for beta version. Table 1 shows test cases for the alpha version and by using these test cases we decide or evaluate these test cases for the beta version. Comparator in our approach identifies the change in the beta version and values, if changed, is placed in changed value list and unchanged values will be placed in the unchanged operations list. For our case study we find what kind of changes have been taken place in beta version.

4.1. Specification after change (Beta version)

Following are the changes that occurred in the modified part of the specification.

4.1.1. *Syntactical change with no impact*

In this example, we have found change in syntax of elements present within segments in case of EDI invoices. But these changes had no impact on semantic. Only syntax rules need to be changed and test cases are upgraded to newer version. If syntax gets changed with deletion of unused segment it does not cause impact on semantics.

4.1.2. Syntactical change with impact of change on semantics

Given below is an example of syntactical change which also has impact on the semantics. Change in syntax changes expected values.

Case 1: In this case, test cases with id 3 and 4 gets obsolete, when rule sets change with deletion of segment name PER. When a segment is deleted, elements in that segment also gets deleted which means test cases related to all elements become obsolete.

Case 2: If rules for calculating charges, balance etc., changes then test cases are placed in re-testable test cases. For e.g. Instead of addition of three elements previously, addition of only two elements is required in newer version; test cases are placed under re-testable list.

Case 3: If new segments are added, we upgrade our xml file with new test cases.

4.2. Implementation

We have developed a framework to illustrate our proposed approach. Our framework loads invoice into the database through invoice reader. Syntax of segments as well as semantic, which is stored in XML file are parsed. After parsing, this information is passed to comparator. Our module compares alpha and beta version outputs after an invoice is loaded through beta version invoice readers. Comparator compares those values and if the values do not match, test case selector [1] module places those test cases in re-testable list. If some attributes are not compared from test cases, those are placed in obsolete list. Correct comparisons are placed in re-usable test cases.

In a test run there are three steps in execution in JUnit framework (1)SetUp ; where invoice is loaded (2) Execute ; where test scenarios are executed and compares the actual value with an expected value (3) TearDown ; where the loaded invoice gets deleted from the database.

SetUp: This step involves loading of an invoice through invoice reader into the database. An invoice reader contains rules which are applied on invoice when they are persisted into database.

Execute: Execute will run the validation framework to compare both expected and actual values.

TearDown: Since invoice is loaded for regression testing purpose so it is cleaned from database. This step is expensive as it considers integrity constraints and dependency among various tables present in database. Values are deleted by firing SQL queries in database.

4.3. Results

The validation framework was run on telecom dataset. Database used was oracle 11g and time taken to automate 220 test cases using jdk 1.6.0 in fourth release of validation framework is as shown in Table 2. Test cases were put in test suite and run using JUnit 3 [9].

Table 2 Results Of framework

Application	Test case automated	Execution time with our framework	Regression Frequency
Telecom	220	62.456 min	Twice a week

Test Suite loaded invoice of five vendors and validated five invoice readers in its fourth release. Invoice was loaded and after validating the results, invoice was deleted from database.

Table 3 Test Case for Beta version

Test Case ID	Attribute	Reusable	Re-Testable	Obsolete
1	Invoice Date	✓		
2	Invoice Number	✓		
3	Communication Number			✓
4	Communication Number Qualifier			✓
5	Quantity	✓		
6	Identification Code Qualifier	✓		
7	Current balance		✓	

After comparing alpha version results with beta version, our validation framework identified test cases for beta version .Test cases were categorized in three categories as shown in Table 3 i.e., re-usable, re-testable and obsolete. Since test case 1, 2, 5 and 6 produced same result as before; these test cases were placed in list of reusable test cases. Segment related to test case 3 and 4 have already been deleted and thus placed in list of obsolete test cases. Rule for calculating current balance changed and thus test cases related to that rule have been placed under Re-Testable Test Cases.

Table 4 Test Coverage using Validation Framework

Vendor	Class Coverage (%)	Method Coverage (%)	Regression Detected
A	90	57	1
B	83	58	0
C	69	62	0
D	60	56	1

Table 4 gives an overview of class coverage and method coverage through fourth release of validation framework from different vendors. Total no of test cases executed were 220 and two regression were detected.

Referring fig 4.1 (a), we aimed to validate those invoice readers for which tickets were raised more frequently. Total no of LOC detected for four invoice readers were 17,296. With increase in number of unit test cases in every release our code coverage reached to 50% with presence of 5% of dead code as well.

The framework was run using ANT [10] and Emma [11] report was generated which gave an overview of Lines of code covered. Initially in our first release we had 60 test cases. Line of Code coverage increased with an increase in number of test cases and these test cases were further used to validate invoice readers of other vendors also.

Fig. 4.1 (b) shows the variation of time taken in manual testing and automated testing. For manual testing, a developer having complete knowledge of the product, checks manipulation through user interface. As the number of test cases increases manual time also increases. With the use of validation framework developer can save time.

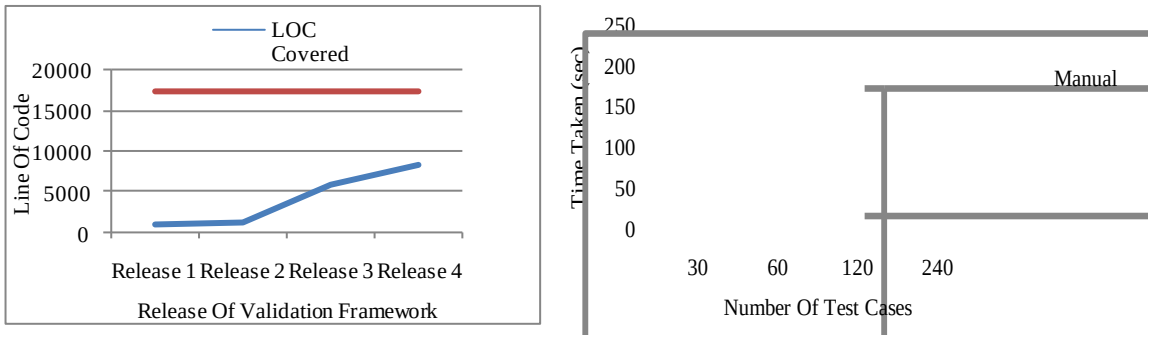

Fig. 4.1 (a) LOC coverage with respect to Release (b) Comparison of time taken by Validation Framework and manual testing

V. ADVANTAGES

The framework helps in detecting regression and this kind of framework can be used on all applications where the rules keep on modifying frequently. The framework can be run on top of any application after new build is passed to ensure the correct working of the results. This kind of framework can reduce manual effort of developers to large extent.

VI. CONCLUSION

In this paper, we have developed a framework for validating the rules of invoices. Vendor format keeps on changing and so it is important for developer to ensure its correctness. Besides, it is equally necessary to check the syntax on invoice format as well. Invoice format should comply with the syntax as well as semantics of guidelines followed by industries to avoid hampering customer's billing calculation. By using the above proposed approach, customer related issue can be resolved without wasting many resources i.e. time, cost and money. This framework was built especially for development team with an aim to let developers know the regression caused by modification done by them in invoice reader module. The proposed approach gives better results in earlier stage of development and saves cost and money. This approach is simple and straightforward and can be embedded with other domains in practice also. In future work, we are extending more test cases to our test suite of alpha version to achieve better code coverage and looking forward for best ways to prioritize test case to save resources.

REFERENCES

[1] Zahid Hussain Qaisar, Shafiq Ur Rehman, A safe regression testing approach for safety critical systems, Advances in Engineering Software 42 (2011), 586–594.
[2] Katarina Grolinger, Miriam A.M. Capretz, A unit test approach for database schema evolution, In Information and Software Technology 53 (2011), 159–170.
[3] Emelie Engström, Per Runeson , Mats Skoglund, A systematic review on regression test selection techniques, Information and Software Technology 52 (2010), 14-30.
[4] Yildiray Kabak , Asuman Dogac, A Survey and Analysis of Electronic Business Document Standards, ACM Computing Surveys, Vol. 42, No. 3, Article 11, Publication date: March 2010.

[5] Florian Haftmann, Donald Kossmann, Eric Lo, A framework for efficient regression tests on database applications, The VLDB Journal (2007) 16:145–164 DOI 10.1007/s00778-006-0028-8.
[6] The Business Rules Group, Defining Business Rules – What Are They Really? rev. 1.3, July, 2000 available online at http://www.BusinessRulesGroup.org.
[7] Yuejian Li,Nancy J.Wahl, An Overview of Regression Testing, In Software Engineering Notes vol 24 no 1.ACM SIGSOFT. January 1999, Page 69.
[8] Dan Parnas & Oleg Test, How Telecom Invoice Automation Can Reduce Costs, In Epiphany Supply Chain Solutions, Inc.
[9] http://junit.org/
[10] http://ant.apache.org/
[11] http://emma.sourceforge.net/

