DEFECT REMOVAL TECHNIQUES FOR REXX

PAT MEEHAN AND PAUL HEANEY

Defect Removal Techniques and their Effectiveness for REXX Applications

Patrick A Meehan IBM IISL PRGS Lab, Dublin, Ireland

Paul Heaney DELPHI Software Limited, Dublin, Ireland

Abstract

Major focus has been put on the reliability of software within the last few years resulting in various attempts to improve that reliability and to produce software with close to zero-defect (six-sigma). Little effort has been expended to measure the relative effectiveness of the different techniques in a controlled fashion.

This paper focuses on the experiences of defect removal of a component of an existing REXX product and the subsequent comparison in a more controlled fashion between different methods of defect removal for a new REXX project.

The main focus of the paper will be on the measured effectiveness of different defect removal techniques and on their suitability to an application that has already been or will be developed in the REXX language with the overall objective of producing close to zero defect REXX applications.

Introduction

Many different philosophies exist as to the best way of ensuring high reliability software systems. Inspections and/or reviews of the different development phases, various forms of code testing and standards by which the development should proceed often figure among these approaches. There has however been little attempt to measure the effectiveness of the different techniques in a controlled fashion.

This paper describes work undertaken by the authors and other participants in an attempt to measure the effectiveness of different defect removal techniques during the coding phase. The incentive to carry out this research was based on our experience with the development of a Program Product component. This work involved the development of key performance changes which varied in complexity from basic performance changes to complex network changes. There were a number of key elements in this development effort.

Performance changes were prototyped at an early stage of the design process to gain some early measurements on their benefit. The resultant code was subjected to some extensive unit testing. Parallel reviews of the entire code were conducted. Results from the reviews were carefully analysed and in some instances the subject code was seeded in an attempt to measure the effectiveness of the parallel reviews. In more complex parts of the development, informal verification by the owner of the code was carried out. The performance component has handed over to formal Test phase with a defect residue of 2.6 defects per KLOC. This low residue compared very well to other components and was less than the average defect residue for projects developed with the Cleanroom techniques (1). The defects that were discovered during formal test were typically of a trivial nature and were easy to fix.

These results suggested that the techniques or at least some of the techniques practiced were very successful. However, it wasn't clear as to which was more effective and whether some combination of

the techniques might be even more effective than others. In order to determine their effectiveness, it was necessary to set up an experiment and measure their efficiencies in a more controlled fashion.

The objective of the experiment was to measure a selection of different techniques on a piece of subject code under a variety of different metrics. The selection was based on techniques typically practiced in software development and are described below.

A small REXX project to manage the reporting of PTR (Problem Tracking Reports) was designed based on well known requirements, the resultant design was reviewed and the code was developed (3K). The resultant code then became the subject of the experiment.

Different Methods

A number of different techniques were employed in order to establish their effectiveness in removing defects from the established REXX program. The exact same REXX code was the subject of all the techniques selected. The following example (please refer to Figure 1 on page 3) which is a selected piece of code from the developed REXX reporting project serves to explain the different methods used and the manner of their use from a REXX perspective.

The inputs to all the techniques were :

- Source code
- Intended function
- Design document

For the purposes of easy reference, each decision with the section of code is referenced on the right hand side of the decision (e.g. B.2.3).

Unit Testing

Unit testing can take on many different forms from the basic statement coverage to the more rigorous form of multiple-condition based unit testing and can vary significantly in their success rates (2).

Decision based Unit Testing

The purpose of this form of testing was to ensure that each decision within the code took on a true and false outcome and then checking that the result was valid. This was carried out by someone other than the code author but who was involved in the original design.

The SIGNAL ON NOVALUE and SIGNAL ON SYNTAX instructions were also added to the code in order to detect uninitialised variables and interpretation errors and NOVALUE and SYNTAX routines were inserted to trap these errors.

In general, where there are n decisions then this would mean 2*n number of test scenarios. However, the number of actual test cases is usually less than this because the different decisions are typically not all independent of each other and even where they are independent of each other they can sometimes co-exist within the same test case.

From the example in the figure (please refer to Figure 1 on page 3), there are 4 decisions. In order for each decision to take on a false and true outcome this would have required the following set of 8 potential test scenarios.

1. b.2, b.2′, b.2.1, b.2.1′, b.2.2, b.2.2′, b.2.3, b.2.3′

where the prime indicates the false outcome of the decisions. On closer examination, it becomes apparent that all of the test scenarios of the form b.2.x′ can be satisfied by the scenario b.2.y where y $\neg$ = x. In addition b.2 must co-exist with any of the list of b.2.1, b.2.2 and b.2.3 so it doesn't have to exist as a separate test case.

So we are really left with the following set of 4 test cases:

1. b.2′
2. b.2.1, b.2
3. b.2.2, b.2
4. b.2.3, b.2

This set of test cases discovered 2 defects in the selected piece of code where the keywords *SUBTRACT* and *ADD* were not included in quotes. It's of interest to note that these would equally have been found through the use of the SIGNAL ON NOVALUE instruction.

```
MANAGE_PCFRAISE:
array = 'RAISE'

 Select
  When type = 'ACTED'  then do
/* Valid Acted and either it was previously OPENed or it was ACTED on..*/
/* but the REL info is different or it was logged as Rejected.....*/
/* Decision..... B.2 */
   If rel ¬ = 'NOT',
     & ('WORD'(p.ptr_no,1) = 'OPEN',
     | ('WORD'(p.ptr_no,1) = 'ACTED' & 'WORD'(p.ptr_no,3) ¬ = rel)),
     | ('WORD'(p.ptr_no,1) = 'REJECT'),
     then do
/* Decision..... B.2.1*/
       If 'WORD'(p.ptr_no,1) = 'OPEN' then
          Call addsub_operator 'WORD'(p.ptr_no,4) 2 'SUBTRACT' array
/* Decision..... B.2.2*/
       If 'WORD'(p.ptr_no,1) = 'REJECT' then
          Call addsub_operator 'WORD'(p.ptr_no,4) 6 'SUBTRACT' array
/* Decision..... B.2.3*/
       If 'WORD'(p.ptr_no,1) = 'ACTED' then
          Call addsub_operator 'WORD'(p.ptr_no,4) 5 SUBTRACT array
       Call addsub_operator ymd_open 3 ADD array
       End

    Otherwise nop
    End

/*.....contd.....*/
```

Figure 1. Sample of Subject REXX Code - Input to all techniques.

Multiple Condition Based Unit Testing

Typically, the code author would unit test his/her code and for this reason the subject code was unit tested by the code author along the lines of multiple condition based unit testing.

Whereas decision based unit testing just focuses on the decision outcome, multiple condition based unit testing focuses on the actual conditions within the decision by ensuring that all possible condition combinations within a decision are exercised.

For example, decision b.2 has 5 different conditions within it and theoretically there are 2 to the power of 5 test scenarios to cover all condition combinations (32). Decisions b.2.1, b.2.2 and b.2.3 have only 1 condition within each and so are handled in the same fashion as with decision based unit testing.

On closer examination of the 5 conditions with decision b.2 it becomes apparent that only a certain subset are possible anyway. For example, the expression *WORD(p.ptr_no,1)* which we refer to as the PTR Status can have only 1 value at a time. If we name the 3 occurrences of this expression as x2, x3 and x5 then the following are the only 4 valid combinations

(x2,x3',x5'), (x2',x3,x5'), (x2',x3',x5), (x2',x3',x5')

where the prime (') indicates the false outcome of the expression. If we name the other conditions in decision b.2 as x1 and x4 then it is clear that these can take on the following 4 valid combinations

(x1,x4), (x1',x4), (x1',x4') and (x1,x4')

Therefore the total number of test cases becomes 4 times 4 or 16 valid test cases which is only half of the number of original scenarios.

Note: Decisions b.2.1 , b.2.2 and b.2.3 are automatically covered by these test cases and no further test cases are required.

This set of test cases discovered the 2 defects already mentioned under decision based unit testing (Test case x1,x2',x3,x4,x5'). In addition, they uncovered a further defect through the following test case combination of (x1',x2',x3',x4',x5) which actually resulted in condition b.2.2 being executed when in fact this particular section of code should not have been entered at all. The coding error was due to the fact that whenever x5 occurred (PTR Status = 'REJECT') regardless of the other conditions x1,x2,x3 and x4, the underlying code was executed whereas it should only have been executed when x5 *AND* x1 occurred. The error arose because the incorrect placement of parenthesis in the decision (Please refer to Figure 2 on page 6).

This defect was not detected under the decision based testing because of it doesn't embrace the different condition combinations within a decision and underlines the inadequacy of the decision based approach.

Verification of REXX Code

Another form of defect removal which has gained some prominence recently particularly since it forms a significant part of the entire Cleanroom methodology is that of verification. As part of the experiment, code verification was undertaken by the code author. This activity took place some 3 months before the multi-condition based unit testing in order to eliminate any potential bias due to the fact that the same person carried out both activities.

Verification is a means of expressing the function of a manageable section of code in an unambiguous fashion and then exercising some intellectual reasoning about the derived function and the original intended function.

The intended function was documented within the actual code when the code was originally written.

A key part of verification was that the code was not executed. Verification was conducted by establishing the derived function for each main section of code within a Procedure and then cascading towards an derived function for the entire Procedure and ultimately an overall derived function for the entire REXX program. The derived function should be sequence free and loop free because this makes it more understandable and more unlike the original code.

Some people advocate a more formal description of the program function; it is our experience that the choice of description for REXX code depends largely on the nature of the code. The authors believe that is important to describe the derived function in a more conceptual fashion and that it is important to divorce it from the actual code details as much as is possible. The use of lists, matrices, and other mathematical notation were considered invaluable.

The key to verification of developed code is a complete understanding of exactly what the code is doing. It is recommended that even where one may think that they know how a particular REXX construct or operating system command works, one should still consult the relevant documentation to verify that understanding. Once that understanding is established, then it is relatively easy to verify it against the intention.

The example (please refer to Figure 2 on page 6) shown is the derived Program Function for two sections of code in a Procedure (B.1 and B.2).

Note: Any non-obvious notation is described separately in the form of specification functions (not shown here).

Program Function B.2 represents the subject code shown in Figure 1 on page 3. The Program Functions were then analysed against the intended function; the intended function for the entire procedure is shown in Figure 3 on page 5.

In this example, the verification discovered all of the defects mentioned so far. A further defect becomes clear when B.2 and B.1 are examined together. From B.1 it is clear that when a new PTR is OPENed that it is added to the weekid corresponding to its OPEN date. However in B.2, we see that when the PTR was already ACTED on but the release information has subsequently changed, the occurrence is deleted from the weekid corresponding to the ACTED date and not from the weekid corresponding to the OPEN date.

Even though the relevant section of code would have been exercised in both types of unit testing, this defect was not found because the weekid for the test case would have been the same for both the ACTED and OPEN dates; even though this is more probable it wouldn't always be the case. This illustrates the dependence of unit testing on selecting the right data which is made much more difficult particularly (as in this case) where the data in question (ACTED date) is not part of a condition within a decision. If the data had been part of a condition then it is more probable, but not definite, that the defect would have been discovered through unit testing.

```
/*.....INTENDED FUNCTION  for MANAGE_PCFRAISE.....*/

This routine is used to manage an  array which contains all of the
information relating to PTRs raised during each week. This array is
subsequently used to fill the file PCFRAISE TABLE. The rows in the
array should be defined according to the following criteria.

ROW   Information

 2    Number of PTRs that are still OPENed during this particular weekid
 3    Number of PTRs OPENed during this weekid which are now ACTED
 4    Number of PTRs OPENed during this weekid which are now
      CLOSED but not due to an injected fix
 5    Number of PTRs OPENed during this weekid which are now
      CLOSED due to an injected fix
 6    Number of PTRs OPENed during this weekid which are now REJECTed
 7    Total number of PTRs OPENed during this weekid which is
      the same as the sum of rows 2,3,4,5 and 6

The routine should take the new information for a PTR (established
earlier) and ensure that the changes are applied to the existing array
information. This should be done by calling a separate routine,
ADDSUB_OPERATOR, with the correct parameters; these are described in
its Intended Function. The routine ADDSUB_OPERATOR makes
the actual changes. In general, new PTR information can mean that
previous information should be deleted and new information added.
```

Figure 3. Intended Function for Sample Subject REXX Code - Input to all techniques.

Parallel Reviews

A number of REXX developers (3) with a cross-section of REXX and VM experience were requested to review the subject code in parallel with the objective of detecting the maximum number of logic defects. They were provided with the design of the reporting system and would have reviewed the code subject to a set of established REXX and VM coding standards. Apart from this, the reviewers were free to use any other defect detection methods. On the example piece of code (please refer to Figure 1 on page 3), the same 2 defects that were found under decision based unit testing were also found but no other additional defects were found on this section of code.

Reviews typically suffer from a lack of structure and can be undisciplined; they are best described as a type of *black box* activity in the sense that we seldom know how they actually are conducted as this is usually left to the discretion of the reviewer.

Overall Results

The results from the experiment have been analysed on a number of different fronts. The graphs (Figure 4 on page 7) illustrate the results for the four primary metrics. Each defect was classified on

```
/* Derived program Function B1.....*/
(PTR is OPEN)
    (PTR was previously unknown) --> +2 (open date)

/* Derived program Function B2.....*/
(PTR is ACTED)
    (PTR was previously OPEN) -->
        +?3? (open date), -2 (previous open date)
    (PTR was previously ACTED & the Release info has changed) -->
        +?3? (open date), -?3? ( previous acted date )
    (PTR was previously REJECTed) -->
        +?3? (open date), 6 (previous open date)
```

Figure 2. Derived Program Function for Sample Subject REXX Code - Verification output

completion of the entire exercise according to its probability of occurrence, the severity from 1 (high) to 4 (low) and its complexity from 1 (low) to 10(high). In addition, the probability- severity metric was defined as the sum of all the probability severity ratios and the overall complexity number as the sum of all the complexity numbers.

Verification of the REXX has proven to be very successful claiming 61 defects out of a total number of 64 that were detected by all of the techniques. The code reviews were the least successful (11 defects) with the success of the unit testing varying according to the type of unit testing conducted. The probability-severity metric underlines the fact that the verification also tended to detect the more severe and the higher probability type of defects with a value of 10.6 compared to 6.1 for multi-condition unit testing. Even though the difference in the number of defects between these two techniques was 17 this deficit accounted for a probability-severity metric of 4.5. The same trends are evidenced when we look at the complexity number for each technique highlighting the fact that the verification is better at finding the more complex defects. The time taken for each technique showed little variation except for the multi-condition unit testing which took up significantly more time.

It's of some interest to look at the defect breakdown. Each defect was classified under one of 4 headings representing the effect of the defect as follows. (Please refer to the graph Figure 5 on page 8).

- A maximum of 7 defects were due to seeds placed in the code.
- Some defects were as a result of either REXX Novalue or Syntax errors
- Other defects resulted in incorrect message handling
- The remaining defects resulted in incorrect results occurring and are difficult to classify further.

All of the techniques were reasonably successful at locating the Novalue/Syntax and the messaging defects. These would typically be classified in the *easy to find* category. However when you look at the more complex defects which sometimes gave rise to subtly incorrect outcomes, the reviews and then the unit testing and finally the verification were successively more successful at detecting them. Similarly when you look at the success at removing the seeds that were placed in the code, the same trend emerges with verification finding all 7 seeds and reviews only finding one of the seeds.

Finally, the VENN diagram illustrates the uniqueness and commonality of defects across the three most successful techniques. All of the three had some uniqueness varying from 1 for each unit testing type to a significant 15 for verification. There were 21 defects which were common to all of the three techniques.

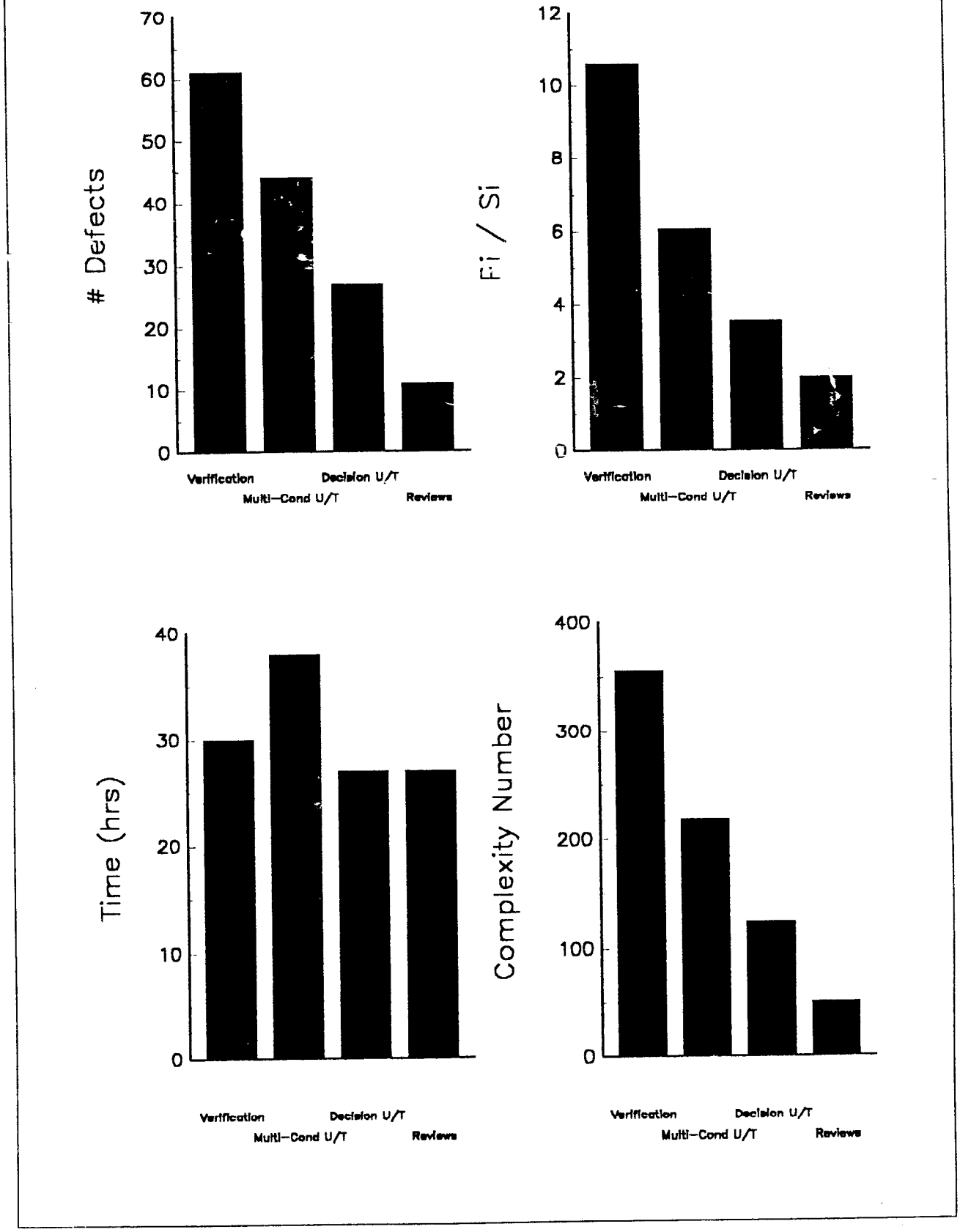
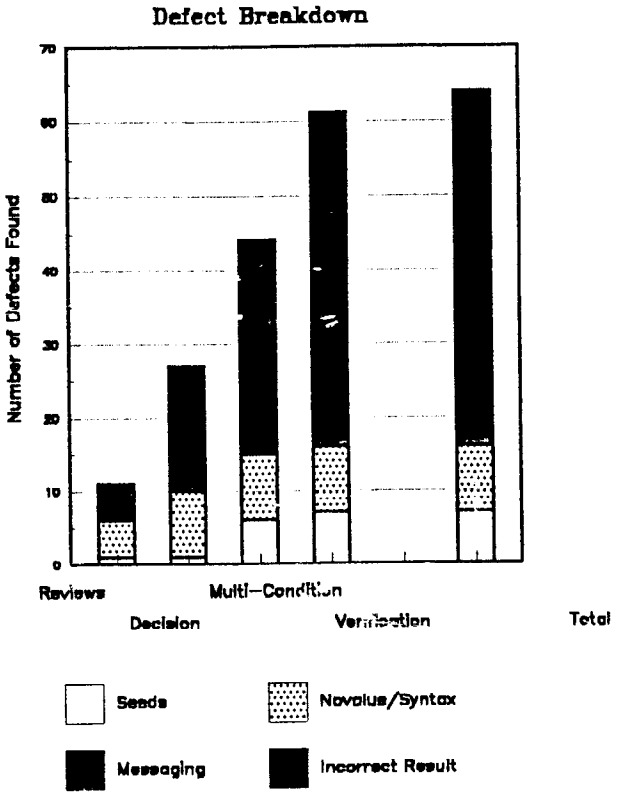


Figure 4. Results of Analysis



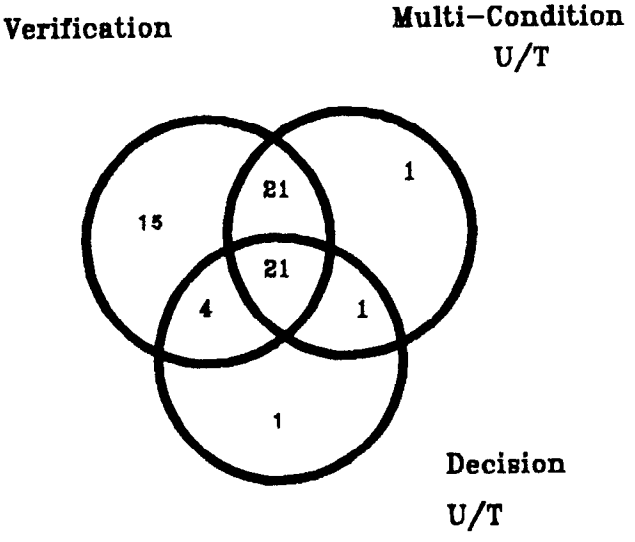


Figure 5. Results of Analysis

Concluding Remarks

Verification should be embraced as a defect removal technique as it has been conclusively shown to be very effective on the subject REXX code and is not as time consuming as expected. REXX as a language is suited to verification in the sense that it is typically easy to understand its different instructions and functions. On the other hand its loose data typing can make it difficult to fully describe the resultant state of the data. For the purposes of subsequent verification, the use of well-structured REXX code makes the task that much easier. Avoidance of REXX flow alterations like ITERATE and LEAVE makes the verification simpler. The verification exercise has also shown the importance of limiting the extent of variables to where they are needed. This can be accomplished easily with the PROCEDURE instruction which protects all existing variables and fully restores them on return from the PROCEDURE. Only those that need to be available can be done through the EXPOSE option. In fact all but one of the defects which were not found by verification were due to the fact that variables were not protected in the fashion described. If one limits the extent of variables as much as possible then the task of defining the program function for the entire program is greatly simplified and the use of the EXPOSE option on all Procedures is an easy way of knowing what variables are not protected.

Even though CMS Pipelines, which implement the pipeline concept under CMS, were not part of the subject code, their use would also appear to benefit the overall verification process in REXX. Pipelines enable complex tasks to be split into small simple robust self contained programs which would be easier to verify.

Even where one still wants to pursue the unit testing path and wants to do it in a rigorous fashion like that described for multi-condition based unit testing, it is our experience (in hindsight) that in fact the derived Program functions which were done as part of verification are an an excellent route to pursue. The program functions typically remove all redundancy, just state the code outcome, are much more understandable than the code itself and hence lend themselves to the task of defining test cases to cover the multi-condition testing rationale.

The other techniques of unit testing and reviews were successively less and less successful. The testing tends to be highly dependent on selecting the right data, cannot satisfactorily deal with missing function and lacks the intellectual control of verification. Reviews are typically black box affairs with the process of carrying out the reviews left largely up to the reviewers and if carried out should be changed to ensure that they embrace verification.

Measurement of the different techniques has provided some invaluable information and shows conclusively the effectiveness of the verification technique and not at the expense of overall productivity.