Abstract: - Often more than 85% of the critical defects in a software product development are introduced into the product in the requirement development process and the product design process, but the existing software testing methods can only be dynamically used after detailed coding. NIST (National Institute of Standards and Technology) concluded that “Briefly, experience in testing software and systems has shown that testing to high degrees of security and reliability is from a practical perspective not possible.” This paper presents a new software testing method called Transparent-Box combining functional testing and structural testing together seamlessly with a capability to automatically establish bidirectional traceability among the related documents and test cases and the corresponding source code according to the test case description. To each test case this method not only helps users check whether the output (if any, can be none when it is dynamically used in requirement development and product design) is the same as what is expected, but also helps users check whether the execution path covers the expected one specified in control flow, so that this method can be used to find functional defects, logic defects, and inconsistency defects. Having an output is no longer a condition to apply this method dynamically, so that it can be used in the entire software development process.

Key-Words: - software, testing, method, software testing, software testing method, quality assurance

1. Introduction - the major existing software testing methods are outdated

Current software quality assurance is mainly based on inspection and functional testing using the Black-Box method[1] that is being applied after the entire product is produced, and structural testing using the White-Box[1] testing method that is being applied after each software unit is coded. It violates Deming’s Product Quality Assurance Principle, “Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.”[2]

Black-box and White-box methods are applied separately without internal logic connections. The White-Box testing is mainly performed in unit testing to test an Existing product rather than a Required product, while the Black-Box testing is mainly performed in system testing, so that both methods and the corresponding techniques and tools cannot be used dynamically in the requirement development process and the software design process where about 85% of critical defects are introduced into a software product as shown in Fig. 1. Even if a requirement development defect or a design defect can be found by both methods after coding, it is too late: the cost for removing the defect may increase tenfold several times.

For those software testing methods, NIST (National Institute of Standards And Technology) concluded that “Briefly, experience in testing software and systems has shown that testing to high degrees of security and reliability is from a practical perspective not possible. Thus, one needs to build security, reliability, and other aspects into the system design itself and perform a security fault analysis on the implementation of the design.” (“Requiring Software Independence in VVSG 2007: STS Recommendations for the TGDC,” November 2006 http://vote.nist.gov/DraftWhitePaperOnSIinVVSG2007-20061120.pdf ).

Those software testing methods and the related techniques and tools are designed to work with the old-established software engineering paradigm based
on linear thinking and the superposition principle that **the whole of a system is the sum of its parts**, so that almost all tasks/activities are performed linearly, partially, locally, and qualitatively, making the defects introduced in upper phases easy to propagate to the lower phases to increase the defect removal cost up to more than 100 times. This old-established software engineering paradigm is entirely outdated, and should be replaced by a new revolutionary software engineering paradigm based on nonlinear thinking and complexity science[3].

2. **The Transparent-Box testing method**

The Transparent-Box testing method is graphically described in Fig. 2.

As shown in Fig. 2, with the Transparent-Box testing method, to each test case, the corresponding tool will not only check whether the output (if any, can be none when it is dynamically used in the requirement development phase and design phase) is the same as what is expected, but also help users to check whether the execution path covers the expected one specified in control flow, and whether the execution hits some modules or branches which are prohibited for the execution of the corresponding test case, plus that it can establish the bi-directional traceability among the related documents and test cases and the source code according to the description of the test case. Having an output is no longer a condition to apply this method, so that it can be used dynamically in the entire software development process for defect prevention and defect propagation prevention.

The bidirectional traceability between test cases and the source code tested is established through the use of Time Tags (when a test case is executed) to be automatically inserted into the descriptions of the test cases and the database of the source code test coverage analysis for mapping them together accurately. Examples of Time Tags that are automatically inserted into the description part of test cases are shown in Fig. 3.

For extending the traceability to include the related documents, some keywords such as @WORD@, @HTML@, @PDF@, @BAT@ are used for indicating the format of the documents and automatically opening the corresponding documents traced at a location specified by a bookmark.

The simple rules for designing a test case description are as follows:

- A ‘#’ character at the beginning position of a line means a comment.
- An empty line separates different test cases.
- Within comments, users can use some keywords such as @WORD@, @HTML@, @PDF@, and @BAT@ to indicate the format of a document, followed by the full path name of the document, and a bookmark.
- Within comments, users can use [path] and [/path] pair to indicate the expected path in three possible ways: module-level path (a list of modules from the entry-module to the end-module), segment-level path (a list of segments from the entry-module to the end-module), and mix module and segment path (combination of partial modules and partial segments from the entry-module to the end-module). When it is applied to a very long path, users may indicate some critical modules or the segments of some critical modules to be covered by the corresponding test case execution.
- Within comments, users can use Expected Output to indicate the expected value to be produced, used for manual or automatic comparison.
- Within comments, users can also use Not_Hit keyword to indicate modules or branches (segments) which are prohibited to enter for the related test case execution.
- After the comment part, there is a line to indicate the directory for running the corresponding program.
- The final line in a test case description is the command line (which may start a program with the GUI) and the options.
An sample test case script file with some test case descriptions is listed as follows (TestScript1):

```
# test case 1 for New Order
# @HTML@ C:\Billing_and_Payment\Requirement_specification.htm#New_Order
# @WORD@ C:\Billing_and_Payment\Prototype_design.doc bmname New_Order
# @WORD@ C:\Billing_and_Payment\TestRequirements.doc bmname New_Order
# [path] main(int, char**) {s0, s1, s9} [/path]
# Expected output : none
C:\Billing_and_Payment
Billing_and_Payment.exe new_order Confirm
```

```
# test case 2 for Pay Invoice
# @HTML@ C:\Billing_and_Payment\Requirement_specification.htm#Pay_Invoice
# @WORD@ C:\Billing_and_Payment\Prototype_design.doc Pay_Invoice
# @BAT@ C:\isa_examples\ganttpro\ganttpr9.bat
# [path] main(int, char**) {s1, s6, s9, }B-Pay_Invoice(void) [/path]
# Expected output : none
C:\Billing_and_Payment
Billing_and_Payment.exe Pay_Invoice
```

About how the segment numbers are assigned for a program module, let us see the following example:

A sample “main(int, char**)” program:

```c
#include <stdio.h>
#include <string.h>
void main(int argc, char** argv)
{
    int ERROR_CODE;
    if(argc != 3 && argc != 4)
        printf("Error found in the command-line.
");
    else if (argc == 3){
        if(strcmp(argv[1],"global_placement")==0)
            // calling g_placement(argv[2]);
        else if(strcmp(argv[1],"global_routine")==0)
            // calling g_routine(argv[2]);
        else if(strcmp(argv[1],"detailed_placement")==0)
            // calling d_placement(argv[2]);
        else if(strcmp(argv[1],"detailed_routing")==0)
            // calling d_routing(argv[2]);
        else if(strcmp(argv[1],"partititionning")==0)
            // calling partitioning(argv[2]);
        else if(strcmp(argv[1],"ordering")==0)
            // calling ordering(argv[2]);
        else
            // calling printf("Invalid name: %s
",argv[1]);
    } else if (strcmp(argv[2],"dbs_build") == 0)
        // calling dbs_build(argv[2],argv[3]);
    else printf("Error! Invalid name: %s
",argv[1]);
}
```

The corresponding segment numbers assigned are shown in Fig. 4 with that the tested segments are shown in red color automatically:

Fig. 5 shows the facility for the establishment of automated and self-maintainable traceability using Time Tags and book marks.

For instance, to C++, an “if” statement will be treated using the “?:” operation to support MC/DC (Modified Condition/Decision) test coverage analysis.

A statement as:

```c
if (a && b) printf ("OK\n");
```

will be changed to:

```c
if (((a) ? (aisai_rp -> con[0] |= excc, 1) : (aisai_rp -> con[1] |= 0x33, 0)) && (b)? (aisai_rp -> con[1] |= excc, 1) : (aisai_rp -> con[2] |=
```
excc, 1) : (aisai_rp -> con[2] | = 0x33, 0))
printf("OK
\n");

Note: the array aisai_rp -> con is used to record the code coverage data for all condition outcomes, not only for the branches.

After test case execution, a relationship table between the test cases (represented by the Time Tags T1, T2...Tn) and the modules can be automatically built as follows (here the number “1” means the module is tested), see Table 1:

Table 1: the relationship between the test cases and the modules of a program being tested

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>M11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>M12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

... ... ... ... ... ... ...

Similarly, another relationship table between the test cases and the code segments of a program module can also be automatically built as shown in Table 2.

Table 2: the relationship between the test cases and the segments of a program module

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>S4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>S5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>S6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>S7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>S8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>S9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>S10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>S11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>S12</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

... ... ... ... ... ... ...

In the implementation, we use one bit rather than one byte to represent the test result of each module and each segment to save needed space greatly.

With those data, we can easily trace the relationship automatically using the test case script window and test coverage window as shown in Fig. 6 to Fig. 8.

The operations for forward tracing – click a test case in the test case script window, the corresponding tool will highlight the selected test case in blue, then the segments and modules that can be tested by the test case will be highlighted in red on the Source Code window according to the Time Tags - see Fig. 6 – 7.

 Fig. 6 An application example of forward traceability established

Fig. 7 Another application example of forward traceability established

The operations for backward tracing – click a segment (or module) on the Source Code window to select it, then the corresponding tool will highlight the selected segment or module in blue in the Source Code window, while the corresponding test cases will be highlighted in red in the Test Case window through the mapping of the Time Tags – see Fig. 8.

 Fig. 8 An application example of backward traceability established
Why traceability is important to software development?

"... Important benefits from traceability can be realized in the following areas: project management, process visibility, verification and validation (V&V), and maintenance [5]:

**Project Management**
Traceability makes project management easier by simplifying project estimates. ...

**Process Visibility**
Traceability offers improved process visibility to both project engineers and customers....

**Verification and Validation**
The most significant benefits provided by traceability can be realized during the V&V stages of a software project. Traceability offers the ability to assess system functionality on a per-requirement basis, from the origin through the testing of each requirement. Properly implemented, traceability can be used to prove that a system complies with its requirements and that they have been implemented correctly. If a requirement can be traced forward to a design artifact, it validates that the requirement has been designed into the system. Likewise, if a requirement can be traced forward to the code, it validates that the requirement was implemented. Similarly, if a requirement can be traced to a test case, it demonstrates that the requirement has been verified through testing. Without traceability, it is impossible to demonstrate that a system has been fully verified and validated.

**Maintenance**
Traceability is also a valuable tool during the maintenance phase of a software project for many of the same reasons that it is valuable for project management. Initially defined requirements for a software project often change even after the project is completed, and it is important to be able to assess the potential impact of these changes. Traceability makes it easy to determine what requirements, design, code, and test cases need to be updated to fulfill a change request made during the software project’s maintenance phase. ”[6]

The major features of the established traceability

The major features include the following:

**Automated**
This facility works automatically with the capability to insert the Time Tags into both the test case description part and the database of the program test coverage measurement result, and highlight the test cases selected on the corresponding test script window, and the source code modules/branches shown in a control flow diagram in the corresponding source code window, or vice versa, as well as open the related documents traced from the locations pointed by the bookmarks.

**Self-maintainable**
This facility is self-maintainable no matter if the contents of a document are modified, the parameters of a test case are modified, or the source code is modified – after rerunning the test case scripts, the traceability will be automatically updated without manual rework.

**Methodology-independent**
This facility is methodology-independent, no matter which methodology or process models are used to develop the product.

**Nonlinear, bidirectional, and parallel**
This facility works in a nonlinear, bidirectional, and parallel style – when a design defect is found after the product delivery, the developers can perform backward tracing to check the related requirement, and forward tracing to find and fix the related source code.

**Accurate**
This facility is based on the dynamic execution of the test cases and test coverage measurement and the time tags to map the test cases and the source code tested, so that it is accurate. After code modification or parameter changes of the test cases, we can re-run the test cases to automatically update the facility.
Precise

This facility is precise to the highest level – up to the code statement/segment (a set of statements to be executed with the same conditions) level, bi-directionally. It is particularly useful for side-effect prevention in software maintenance.

3. The new software testing paradigm based on the Transparent-Box testing method

Based on the Transparent-Box method, a new revolutionary software testing paradigm is established which offers comprehensive functions and capabilities for software testing, including the support for MC/DC (Modified Condition/Decision Coverage) test coverage analysis, memory leak and usage violation check, performance analysis, runtime error type analysis and execution path tracing, GUI operation capture and selective playback, test case efficiency analysis and test case minimization for efficient regression testing after code modification, incremental unit testing and integration testing combined together seamlessly, semi-automatic test case design, and more.

This new software testing method can be applied in the requirement development process for finding logic defects and inconsistency defects efficiently with the Holistic, Actor-Action and Event-Response Driven, Traceable, Visual, and Executable (HAETVE) software requirement development technique innovated by Jay Xiong to be used to replace the Use Case approach (which is not holistic, not suitable for event-response type applications, not traceable, and not executable for defect removal). Application examples are shown in Fig. 9 – Fig. 11.

Fig. 9 An application result of the HAETVE technique for the function decomposition of the functional requirements of a Billing_and_Payment product through stub programming using stub modules (there are some function call statements in the body of a module (or an empty body) without real program logic)

The stub programming source code of the main() module is listed as follows:

```c
void main(int argc,char** argv) {
    int key;
    if(argc==1 /* Missing a parameter */) {
        cout << "Invalid Commands: \n" << argv;
    } else {
        if(strcmp(argv[1],"New_Order")==0 ||
            strcmp(argv[1],"New_order")==0 ||
            strcmp(argv[1],"new_order")==0 ) {
            A_New_Order();
            cout << "\n\n*** A_New_Order () called. ***\n";
        } else if (strcmp(argv[1],"Confirm_Order")==0 ||
            strcmp(argv[1],"Confirm_order")==0 ||
            strcmp(argv[1],"confirm_order")==0 ) {
            C_Confirm_Order();
            cout << "\n\n*** C_Confirm_Order () called. ***\n";
        } else if (strcmp(argv[1],"Invoice_Buyer")==0 ||
            strcmp(argv[1],"Invoice_buyer")==0 ||
            strcmp(argv[1],"Invoice_buyer")==0 ) {
            D_Invoice_Buyer();
            cout << "\n\n*** D_Invoice_Buyer() called. ***\n";
        } else if (strcmp(argv[1],"Pay_Invoice")==0 ||
            strcmp(argv[1],"Pay_invoice")==0 ||
            strcmp(argv[1],"pay_invoice")==0 ) {
            B_Pay_Invoice();
            cout << "\n\n*** B_Pay_Invoice() called. ***\n";
        } else if (strcmp(argv[1],"Send_Reminders")==0 ||
            strcmp(argv[1],"Send_reminders")==0 ||
            strcmp(argv[1],"send_reminders")==0 ) {
            E_Send_Reminders ();
            cout << "\n\n*** E_Send_Reminders () called. ***\n";
        } else cout << "Invalid Commands: \n" << (char**) argv <<endl;
    }
}
```

After the execution of the test script file, TestScript1, using this new software testing paradigm through the Panorama++ product, one logic defect and another inconsistency defect were found as shown in Fig. 10.
After checking the source code, we can easily find that there is a defect coming from an extra space character:

```c
if(argc==1 /* Missing a parameter */ ||
   argc > 2 /* Having an extra parameter */ )
{
    cout << "Invalid Commands: \n" << argv;
}
else
{
    if(strcmp(argv[1],"New_Order")==0 ||
       strcmp(argv[1],"New_order")==0 ||
       strcmp(argv[1],"new_order")==0 )
    {
        A_New_Order();
        cout << "**** A_New_Order () called. ***\n";
    }
}
```

After checking the bookmarks, we found that in the TestRequirements.doc file the bookmark `Now_Oder` is pointing to the Pay Invoice Treatment position rather than the New Order Treatment position. After removing the two defects, a correct result is obtained as shown in Fig. 11.

4. The major features of the new software testing paradigm

The new presented software testing paradigm brings revolutionary changes to software testing. The major features of the new software testing paradigm include:

- It is based on the Transparent-Box testing method which combines functional testing and structural testing together seamlessly with close logic connection and a capability to automatically establish bidirectional traceability among the related documents and test cases and the corresponding source code tested.
- It can be used in the entire software development processes dynamically, from the requirement development process down to the maintenance process.
- It can be used to find functional defects, structural defects, and inconsistency defects.
- It supports MC/DC test coverage analysis required for the RTCA/DO-178B level A [7] standard, being able to show the test coverage analysis results graphically with untested branches and conditions highlighted as shown in Fig. 12.
Why is MC/DC (Modified Condition/Decision Coverage) essential to commercial software products?

Often people believe that statement-level test coverage is not good enough for the quality assurance of commercial software, but branch-level test coverage may meet the quality assurance requirements. Is it true?

Before answering the question, let's see some examples.

`func1` is a C program module with the source code as follows:

```c
int func1 (int a, int b, int c)
{
    if (a && b && (c==1 || c==11 ||
                  c==111 || c==1111 || c==11111))
        return c + c/10 + c/100 + c/1000
    else
        return 0;
}
```

If we consider branch-level test coverage only, then there are two logic paths; but if we consider MC/DC test coverage, there are eight logic paths as shown in Fig. 13.

`func2` is another C program module with the same functionality as `func1` but written in different style without using multiple conditions in a decision statement:

```c
int func2 (int a, int b, int c)
{
    if (a)
    {
        if (b)
        {
            switch (c)
            {
                case 1:
                    return 1;
                case 11:
                    return 12;
                case 111:
                    return 123;
                case 1111:
                    return 1234;
                case 11111:
                    return 12345;
                default:
                    return 0;
            }
        }
    }
    return 0;
}
```

The number of source lines of `func2` is 25, while the number of source lines of `func1` is 8.

The number of logic paths for `func2` is eight too as shown in Fig. 14.

With the presented software testing paradigm, unit testing and integration testing are combined together incrementally according to the bottom-up testing order assigned on the corresponding call graph (an example is shown in Fig. 15) without designing and using stub units in real cases (if a stub unit is used, it will not return the real value).

- It supports memory leak analysis and memory usage violation check. An application example is shown in Fig. 16.
• It supports performance analysis with the capability to report the branch execution frequency to locate performance bottlenecks better as shown in Fig. 17.

Fig. 17 An application example of performance analysis performed by Panorama++

• It supports efficient test case design by automatically choosing a typical path with the most untested branches and automatically extracting the execution conditions of the chosen path as shown in Fig. 18.

Fig. 18 Assisted test case design performed by Panorama++

• It supports embedded software testing too, as shown in Fig. 19.

Fig. 19 An application example shows that the MC/DC test coverage data are sent from the target to the test server

5. A general comparison between the new software testing paradigm and the old one

(a) The defect finding efficiency

The old testing paradigm used for incremental software development is shown in Fig. 20[4].

Fig. 20 Traditional software testing performed with incremental software development

The old testing paradigm used for the iterative software development is shown in Fig. 21.

Fig. 21 The old testing paradigm used for the iterative software development[4]

The presented new software testing paradigm used for incremental or iterative software development is shown in Fig. 22.

Fig. 22 The presented new software testing paradigm used for incremental or iterative software development

Comparing Fig. 20, Fig. 21, and Fig. 22, it is clear that the new software testing paradigm is much more efficient in finding defects in a software product development process.
(b) The timing in finding the defects

The traditional software testing methods can be performed after coding, but it is too late; in comparison, the new presented software testing paradigm can be used in the entire software development processes, including the requirement development process and the design process.

(c) The defect types that can be found

The traditional black-box method can be used to find functional defects; the traditional structural white-box method can be used to find some structural defects for the Existing product no matter if it is the customer-required product or not.

The presented new software testing paradigm can be used to find functional defects, structural defects, logic defects, and inconsistency defects.

Some functional defects cannot be found by the black-box method, but can be found by the new software testing paradigm as shown in Fig. 21.

(d) The graphical representation techniques for displaying the test results

The test results obtained from the applications of most traditional software testing methods and tools are shown in textual formats or value tables. But the test results obtained from the applications of the presented new software testing paradigm is graphically shown in the system-level and in the detailed source code level as shown in Fig. 22.

(e) The capability to support automated traceability

It is only supported by the presented new software testing paradigm.

Conclusion

This paper presented a new software testing paradigm based on the Transparent-Box testing method combining structural testing and functional testing together seamlessly with internal logic connections and a capability to establish bi-directional traceability among the related documents and test cases and the source code, and can be used dynamically in the entire software development processes from requirement development down to maintenance.

References