Software Testing Revolution Based on Complexity Science - An Introduction to NSE Software Testing Paradigm

WanYuan Huang¹, Jay Xiong²

¹The Jumpulse Center of Research and Incubation of Northwestern Polytechnic University
²NSESoftware, LLC., USA

Abstract - This paper presents a new software testing paradigm (NSE software testing paradigm) based on complexity science using 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. To each test case it checks not only whether the output (if any, can be none when it is dynamically used in requirement development phase and design phase) is the same as what is expected, but also helps users to check whether the real execution path covers the expected one specified in the control flow, and whether the execution hits some modules or branches which are prohibited for the execution of the corresponding test case, so that it can be used to find functional defects, logic defects, and inconsistency defects. Having an output is no longer a condition to apply this method, so that it can be used dynamically in the entire software development lifecycle for defect prevention and defect propagation prevention. With NSE (Nonlinear Software Engineering paradigm) software testing is performed nonlinearly, holistically. Globally, and quantitatively in all phases of a software product development.

Keywords: software testing, method, reliability, quality assurance

1. Introduction

The purpose of software testing is to validate/verify whether a software product meets the customers’ needs and the requirement specifications, find and fix bugs to help users increase the reliability of a software product.

Unfortunately, current software testing methods are outcomes of linear thinking, reductionism, and the superposition principle that the whole of a nonlinear system is the sum of its parts, so that with them almost all software testing activities are performed linearly, partially, locally, qualitatively, inefficiently, and blindly such as the regression testing of the implementation of requirement changes or code modifications. For instance, as shown in Fig. 1 most critical software defects are introduced into a software product in the requirement development phase and the design phase, but current dynamic software testing is performed after detailed coding – it is inefficient and too late.

![Fig. 1 Current software testing is inefficiently performed after coding](image)

About 85% of the critical defects are introduced to a software at upstream, but the testing methods can only be used dynamically in downstream

2. The major existing software testing methods, techniques, and tools are outdated

Current software quality assurance is mainly based on (1) functional testing using Black-Box testing method being applied after the entire product is produced; (2) structural testing using White-Box testing method being applied after each software unit is coded; and (3) product review using untraceable documents and untraceable source code. It violates Deming’s Product Quality Assurance Principles that “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.”[1]

Both testing methods are applied separately without internal logic connection. 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 phase and the software design phase. 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 will 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

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, and locally, making the defects introduced in upper phases easy to propagate to the lower phases to increase the defect removal cost 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[2].

3. 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 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, so that it can be used to find functional defects, logic defects, and inconsistency defects. Having an output is no longer a condition to apply this method, so that it can be used dynamically in the entire software development lifecycle 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 to be automatically inserted into the description of the test cases and the database of the source code test coverage analysis for mapping them together accurately. Examples of Time Tags are automatically inserted into the description path of test cases shown in Fig. 3.

For extending the traceability to include the related documents, Some keywords such as @word@, @HTML@, @PDF@, @BAT@ are used for 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 means a separator between 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 for a test case.
Within comments, users can use Expected Output to indicate the expected value to be produced.
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.
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):

```c
# test case 1 for New Order
# @HTML@
C:\Billing_and_Payment10\Requirement_specification.htm#New_Order
# @WORD@
C:\Billing_and_Payment10\Prototype_design.doc bname New_Order
# [path] main(int, char**) {s0, s1, s9} [/path]
# Expected output : none
C:\Billing_and_Payment10 Billing_and_Payment.exe new_order Confirm

# test case 2 for Pay Invoice
#@HTML@
C:\Billing_and_Payment10\Requirement_specification.htm#Pay_Invoice
#@WORD@
C:\Billing_and_Payment10\Prototype_design.doc Pay_Invoice
# [path] main(int, char**) {s1, s6, s9, }B-Pay_Invoice(void) [/path]
# Expected output : none
C:\Billing_and_Payment10 Billing_and_Payment.exe Pay_Invoice
```

4. The new software testing paradigm based on complexity science using the Transparent-Box testing method

Based on complexity science using the Transparent-box method, a new revolutionary software testing paradigm is established which offers comprehensive functions and capabilities for software testing, including the support not only for functional testing, but also for MC/DC (Modified Condition/Decision Coverage) test coverage analysis, memory leak and usage violation check, performance analysis, runtime error type analysis and the 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.

Application examples of this new software testing paradigm in the requirement development phase for finding logic defects and inconsistent defects efficiently with the Holistic, Actor-Action and Event-Response Driven, Visual, Traceable, 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 directly executable for defect removal) are shown in Fig. 3 to Fig. 4.

![Function Call Graph](image)

**Fig. 4** An application result of the HAETVE technique for the function decomposition of the functional requirements of a Billing_and_Payment product through dummy programming using dummy modules (there are some function call statements in the body of a module (or an empty body) without real program logic)

The dummy 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 */
    || 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";
    } else if (strcmp(argv[1],"Confirm_Order")==0 ||
        strcmp(argv[1],"Confirm_order")==0
        || strcmp(argv[1],"confirm_order")==0 ) {
        C_Confirm_Order();
        cout << "*** 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 << "*** 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 *** 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 *** E_Send_Reminders() called. ***\n";
    } else {
        cout << "Invalid Commands: \n" << argv;
        argv <<endl;
        cout << "*** Executed. *** \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. 5.

After checking the source code, we can easily find that there is a defect coming from an extra space character:

An extra space character is added:

```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";
        } else if (strcmp(argv[1],"Confirm_Order")==0 ||
            strcmp(argv[1],"Confirm_order")==0
            || strcmp(argv[1],"confirm_order")==0 ) {
            C_Confirm_Order();
            cout << "*** 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 << "*** 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 *** 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 *** E_Send_Reminders() called. ***\n";
        } else {
            cout << "Invalid Commands: \n" << argv;
            argv <<endl;
            cout << "*** Executed. *** \n" << (char**) argv
            <<endl;
        }
    }
```

After checking the bookmarks, we found that in the TestRequirements.doc file the bookmark New_Order 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. 6.
When this new software testing paradigm is applied to test a software program without the source code, we can design a virtual main() to indicate the corresponding operations and call the program indirectly through dummy programming too. In this way the GUI operation can be captured and automatically play back after code modification with the capability to establish bidirectional traceability to find the inconsistency defects among the test cases, the test requirements, and user’s manual, and other related documents even if the source code is not available.

5. 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, as shown from Fig. 3 to Fig. 5.
- It can be used in the entire software development lifecycle dynamically, from the requirement development phase down to the maintenance phase.
- It can be used to find functional defects, structural defects, inconsistency defects, memory leaks and memory usage violation defects, and performance bottlenecks.
- It supports MC/DC test coverage analysis required for the RTCA/DO-178B level A standard, being able to show the test coverage analysis results graphically with untested branches and conditions highlighted as shown in Fig. 7.
- It supports memory leak analysis and memory usage violation check. An application example is shown in Fig. 8.
- It supports performance analysis with the capability to report the branch execution frequency to locate performance bottlenecks better as shown in Fig. 8.
- 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. 9.

![Assisted test case design performed by Panorama++](image)

Fig. 9 Assisted test case design performed by Panorama++

- It supports embedded software testing too, as shown in Fig. 10.

![An application example shows that the MC/DC test coverage data are sent from the target to the test server](image)

Fig. 10 An application example shows that the MC/DC test coverage data are sent from the target to the test server

6. 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. 11[3].

![Traditional software testing performed with incremental software development](image)

Fig. 11 Traditional software testing performed with incremental software development

The old testing paradigm used for the iterative software development is shown in Fig. 12.

![The old testing paradigm used for the iterative software development](image)

Fig. 12 The old testing paradigm used for the iterative software development

The presented new software testing paradigm used for incremental or iterative software development is shown in Fig. 13.

![The presented new software testing paradigm used for incremental or iterative software development](image)

Fig. 13 The presented new software testing paradigm used for incremental or iterative software development

Comparing Fig. 11, Fig. 12, and Fig. 13, 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 all phases of a software development lifecycle, including the requirement development phase and the design phase.

(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 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 can not be found by the black-box method, but can be found by the new software testing paradigm as shown in Fig. 14.

(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. 15.

(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 which brings revolutionary changes to software testing in the 21st century by combining structural testing and functional testing together seamlessly with internal logic connections, which can be used dynamically in the entire software development lifecycle from requirement development down to maintenance.

**References**

