Transparent-Box: Efficient Software Testing Method Combining Structural and Functional Testing together

Jay Xiong, Lin Li

Abstract**—** Existing software testing methods cannot be dynamically used in requirement modeling and system design before detailed coding. But often, more than 85% of the critical defects in a software product development are introduced into the product in the requirement modeling process and the product design process. Therefore, it is easy to understand why 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 dynamically used in the entire software development process from the first place down to the retirement of a software product to find functional defects, logic defects, and inconsistency defects.

Keywords**— software, testing, method, software testing, software testing method, quality assurance**

I. Introduction: The Major Existing Software Testing Methods Are Outdated

Software testing (ST) is the process of identifying and delivering the software as a product based on the specification that has been given and required by the users[1]. Software testing is mainly using the Black-Box method[2] that is being applied after the entire product is produced, and White-Box[2] testing method that is being applied after each software unit is coded. 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.

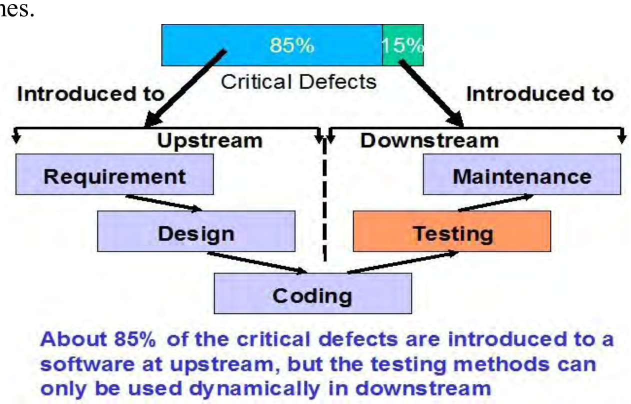


Fig. 1 Current software testing methods cannot be dynamically used in upstream of software engineering

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

Jay Xiong was with The Acadamy of Science of China, Hitachi in Japan, and U.C. Berkeley.. Now he is with NSEsoftware, LLC. USA (email: jay@nsesoftware.com) .
Lin Li is with NSEsoftware, LLC., USA (email: lilin@nsesoftware.com)

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].

II. THE TRANSPARENT-BOX TESTING METHOD

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

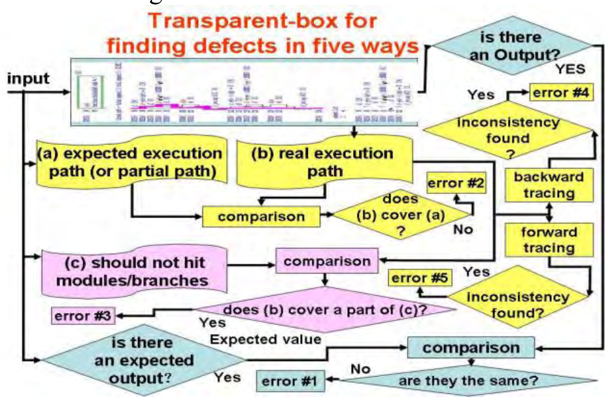


Fig. 2 Transparent-Box testing method

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.

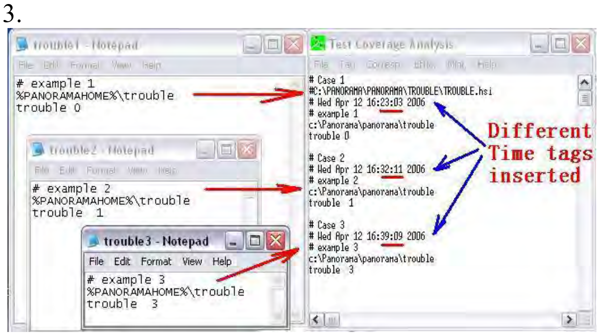


Fig. 3 Time Tag Examples

For extending the traceability to include the related documents, some keywords such as @WORD@, @HTML@, @PDF@, @EXCEL@, and @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@, @EXCEL@, 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_Payment10\Requirement_specification.htm#New_Order
# @WORD@ C:\Billing_and_Payment10\Prototype_design.doc bmname New_Order
# @WORD@ C:\Billing_and_Payment10\TestRequirements.doc bmname 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\Protorype_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_Payment10
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:

```
#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.\n");
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_routing(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],"partitionning")==0)
 ; // calling       partitioning(argv[2]);
else if(strcmp(argv[1],"ordering")==0)
 ; // calling   ordering(argv[2]);
else
 ; // calling   printf("Invalid name:
%s\n",argv[1]);
} else if (strcmp(argv[2],"dbs_build") == 0)
 ; // calling   dbs_build(argv[2],argv[3]);
else printf("Error! Invalid name: %s\n",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.

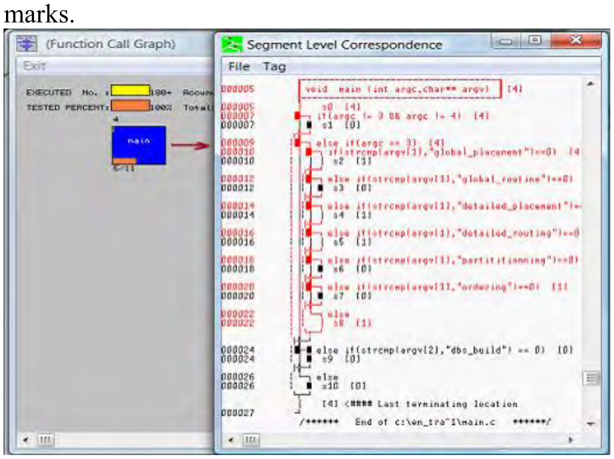


Fig. 4 The control flow of the main() program with segment numbers (s0, s1, s2...) assigned

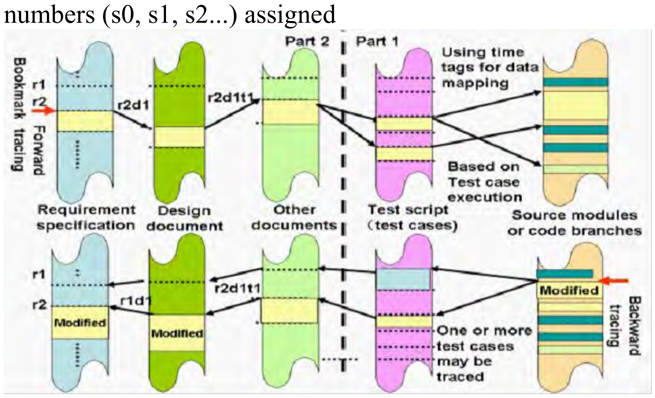


Fig. 5 The facility for automated and self-maintainable traceability

The major steps for establishing and applying the bidirectional traceability are as follows:

Step 1: Organize the requirement specification and the related documents hierarchically with the bookmarks, clearly indicate each requirement and the corresponding test scripts and the test case numbers;

Step 2: Design the test case scripts with the corresponding keywords to indicate the formats and the file paths and the bookmarks for the related documents;

Step 3: Perform code instrumentation for test coverage analysis to the entire program;

Step 4: Compile the instrumented program;

Step 5: Execute the test case scripts with the corresponding tool.

Step 6: Show the modified test case script files with inserted time tags in a window;

Step 7: Show the program test coverage measurement result using a control flow diagram in another window;

Step 8: Perform forward tracing from a test case with a tool to map and highlight the corresponding modules and code branches tested by the test case through the inserted time tag – at the same time, open the related documents according to the document formats, file paths, as well as the bookmarks (or run the corresponding batch file if a @BAT@ keyword is used);

Step 9: Perform backward tracing from a program module or code branch with a tool to map and highlight the related test cases though the inserted time tags – at the same time, open the related documents according to the document formats, file paths, as well as the bookmarks (or run the corresponding batch tile if a @BAT@ keyword is used);

Step 10：After the implementation of code modifications, go to step 3.

Step 11: If a related document is modified in the contents only without changing the bookmarks, there is nothing to do; but if the bookmarks are modified (such as the name of a bookmark is changed), modify the corresponding test case scripts according to the new bookmarks, then go to step 5;

Step 12: If only the test cases are modified, go to step 5;

Step 13: If the source code is modified, go to step 3;

Step 14: If it is the time to perform requirement validation and verification (V&V), use the document hierarchy information organized in step 1 to get each requirement and the corresponding test cases to perform forward tracing one by one to see whether the requirement is completely implemented;

Step 15: If a requirement needs to be modified: (1) get the test cases related to this requirement to perform forward tracing to locate the documents that need to be updated, and the source modules or branches that need to be modified; (2) perform backward tracing from those modules or branches to see whether more requirements are related – if it is related to more requirements, the implementation of the code modification must satisfy all of the related requirements to avoid requirement conflicts.

Step 16: If it is the time to perform regression testing after code modification, get the modified modules or branches to perform backward tracing to collect the corresponding test cases which can be used to re-test the modified program efficiently. Sometimes, there may be a need to add new test cases.

The code instrumentation method used for test coverage analysis are different for different programming languages.

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:

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

will be changed to:

```
if (((a) ? (aisai_rp -> con[0] |= excc, 1) : (aisai_rp -> con[0] |= 0x33, 0)) && ((b)? (aisai_rp -> con[1] |= excc, 1) : (aisai_rp -> con[1] |= 0x33, 0)) ? (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

	T1	T2	T3	T4	T5	T6	T7	...
M1	0	0	0	1	0	0	0	...
M2	1	1	0	0	0	0	1	...
M3	0	1	0	1	0	0	1	...
M4	1	0	0	0	0	0	0	...
M5	0	1	0	1	0	0	0	...
M6	1	0	0	0	0	0	0	...
M7	0	0	0	0	0	0	1	...
M8	0	0	0	0	0	0	0	...
M9	0	0	0	0	0	0	0	...
M10	0	1	0	0	0	0	0	...
M11	1	1	0	1	0	0	1	...
M12	0	0	0	1	0	0	0	...
...	...	...	...	...	...	...	...	...

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

	T1	T2	T3	T4	T5	T6	T7	…
S1	1	1	0	0	0	0	1	…
S2	0	0	0	1	0	0	0	…
S3	1	0	0	0	0	0	0	…
S4	0	1	0	1	0	0	1	…
S5	0	1	0	1	0	0	0	…
S6	1	0	0	0	0	0	1	…
S7	1	1	0	1	0	0	1	…
S8	0	0	0	0	0	0	0	…
S9	0	0	0	0	0	0	0	…
S10	1	1	0	1	0	0	1	…
S11	0	1	0	0	0	0	0	…
...	...	...	...	...	...	...	...	…

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.

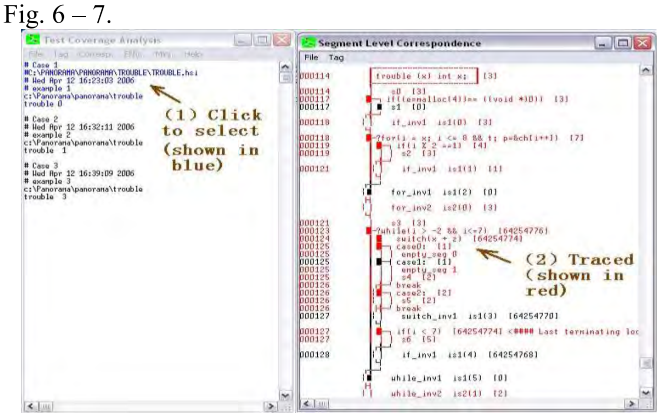


Fig. 6 An application example of forward traceability established

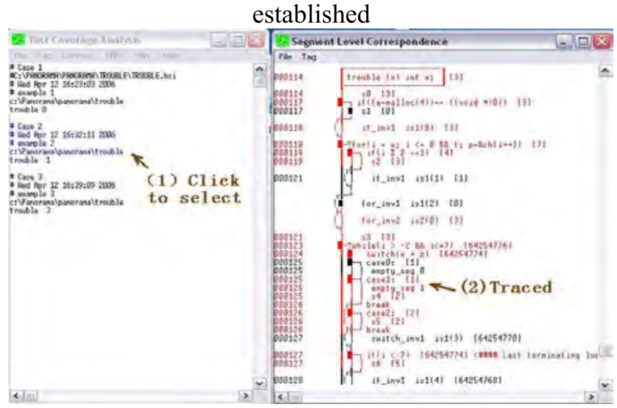


Fig. 7 Another application example of forward traceability established

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.

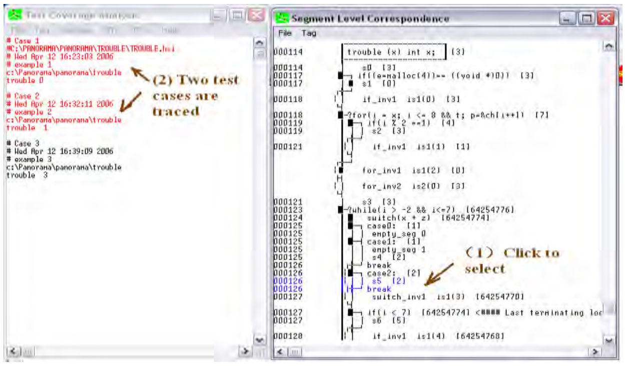


Fig. 8 An application example of backward traceability established

Why is traceability 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 [4]:

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

Software verification and validation include a set of procedures, activities, techniques and tools used in parallel to software development, for ensuring that the product solves the problem that was designed for [5]. 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.

Application examples:

(a) Load the database in the customer site with NSE-Panorama-APL, see Fig. 9.

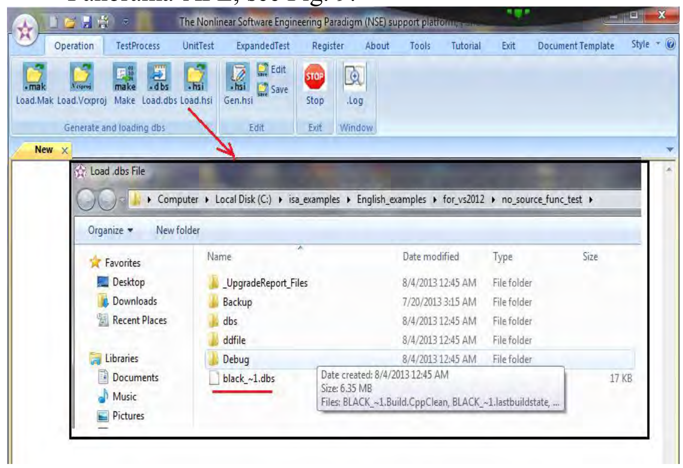


Fig. 9 The process to load the database of an example program

(b) Use the validation tool to load the test database (see Fig. 10).

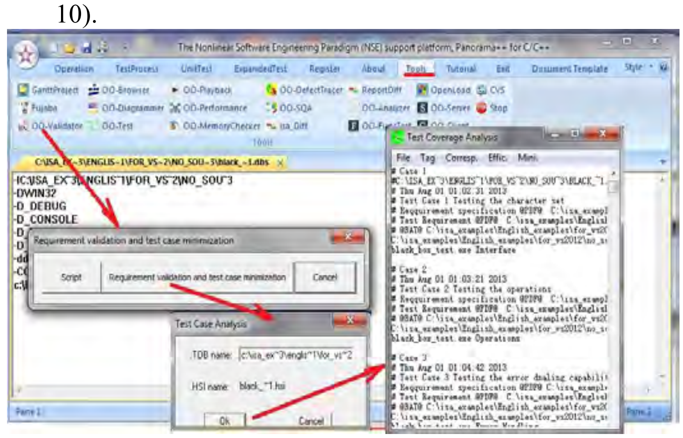


Fig. 10 Load the test database

(c) Open the corresponding control flow windows, see Fig. 11.

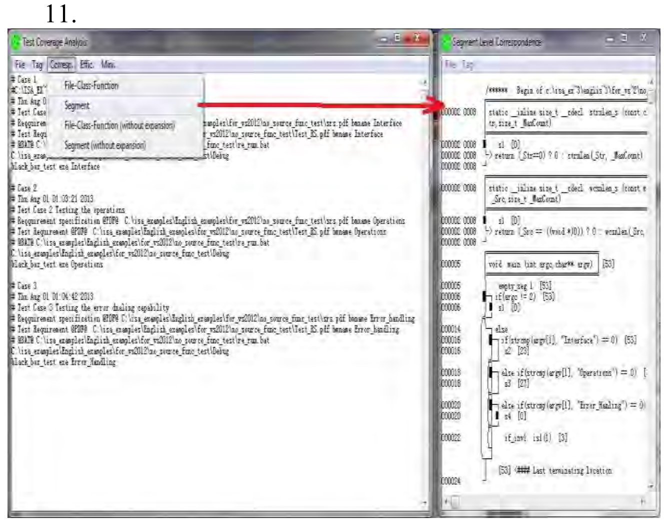


Fig. 11 Open the control flow window

(d) Perform forward tracing to validate the "Interface" design requirement, see Fig. 12.

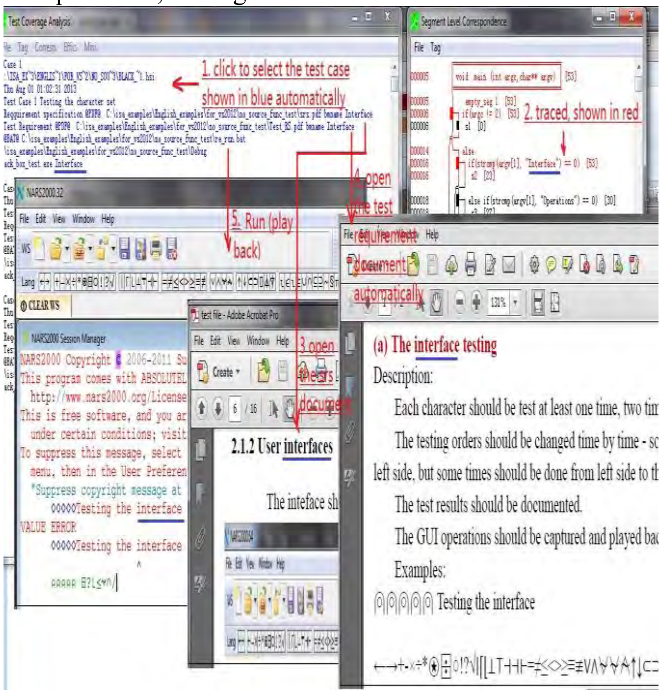


Fig. 12 Perform forward tracing through the corresponding test case to validate the "Interface" requirement (the result shows that it has been implemented correctly – see what have been highlighted in blue lines)

(e) Perform backward tracing through a code branch to validate the "Operation" requirement (the result shows that the requirement has been implemented correctly), see Fig. 13 .

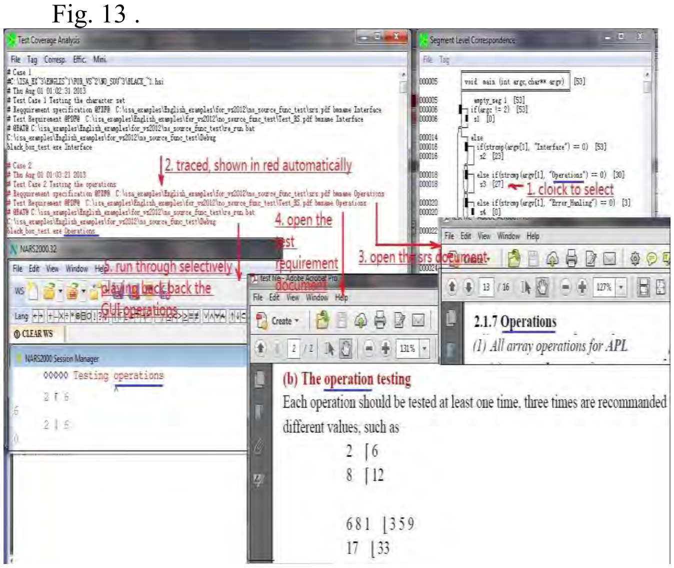


Fig. 13 Backward tracing through a code branch to validate the "Operation" requirement

(f) Found an error through backward tracing: a typing error, see Fig. 14.

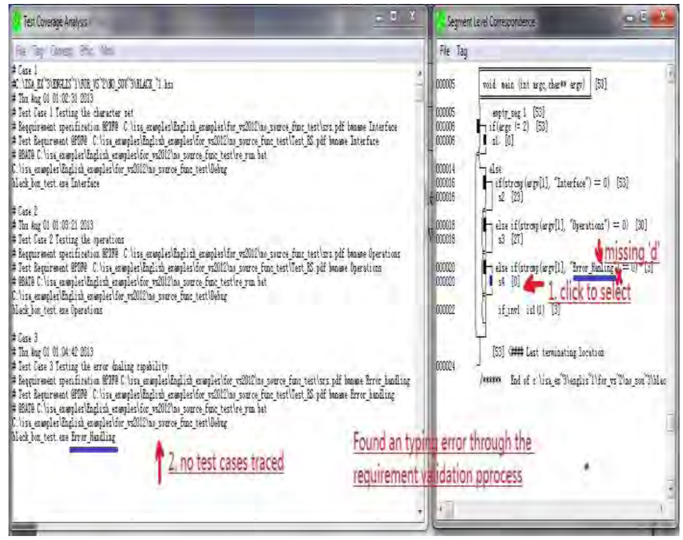


Fig. 14 Found an error through requirement validation and acceptance testing (no test case response)

Conclusion: With the traceability established, NSE-Panorama-APL Acceptance Testing and Requirement Validation Robot will be very useful for automatic and dynamic software requirement validation and acceptance testing.

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.

III. 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. 15 – Fig. 17.

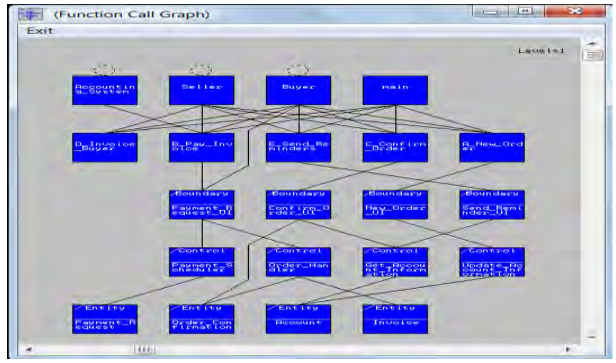


Fig. 15 An application result of the HAETVE technique for the function decomposition of the functional requirements of a Billing_and_Payment product through stub programming using stub modules (there are some function call statements in the body of a module (or an empty body) without real program logic)

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

```
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" << (char**) 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. 16.

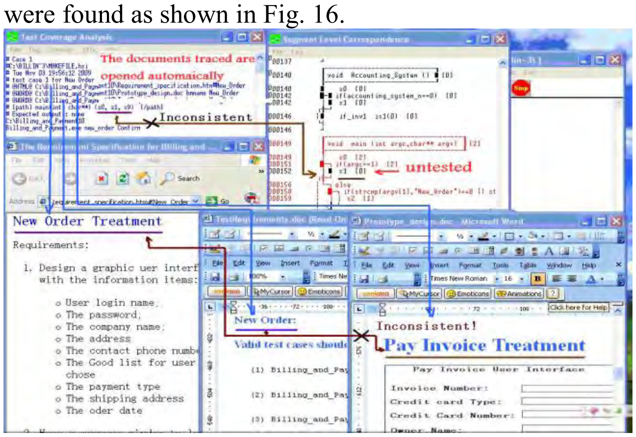


Fig. 16 Two defects found through dynamic testing

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

```
An extra space character found         |
                                        V
  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. 17.

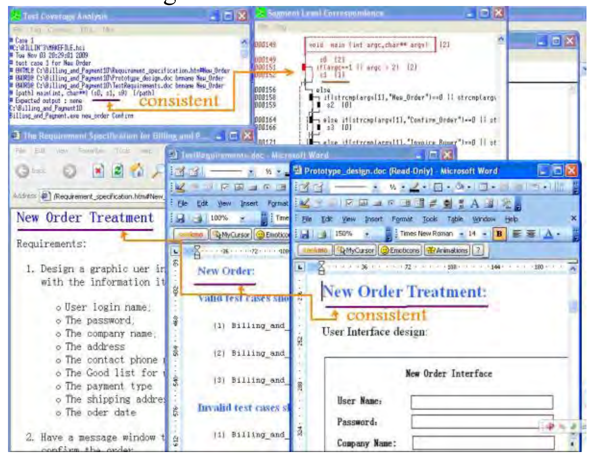


Fig. 17 After modification, the two defects shown in Fig. 4 are removed

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 stub programming too. In this way the GUI operation can be captured and automatically played 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.

IV. 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. 18.

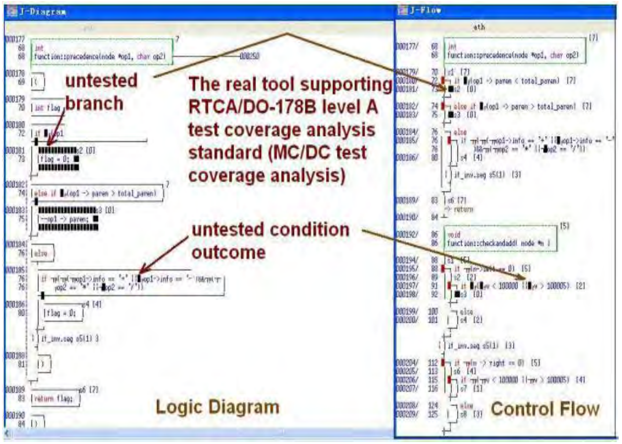


Fig. 18 MC/DC test coverage analysis and the analysis results shown graphically

Why is MC/DC (Modified Condition/Decison 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:

```
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
+ c/10000;
      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. 19.

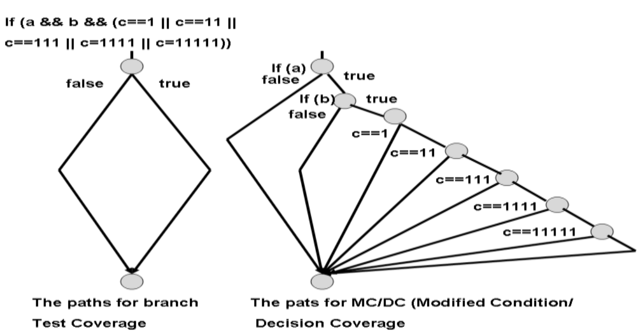


Fig. 19 The logic paths of the func1 program module

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:

```
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. 20.

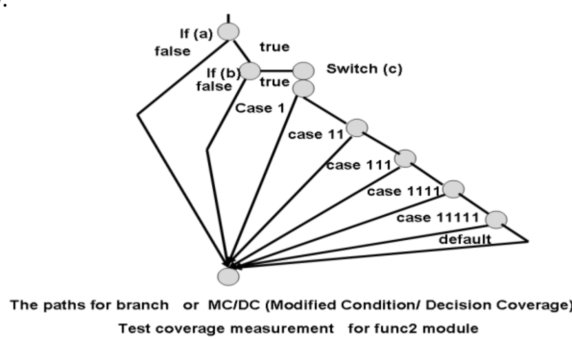


Fig. 20 The number of logic paths of func2 program module

In a software development project, conducting a unit test is an important task but not an easy process[8]. 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. 21) without designing and using stub units in real cases (if a stub unit is used, it will not return the real value) .

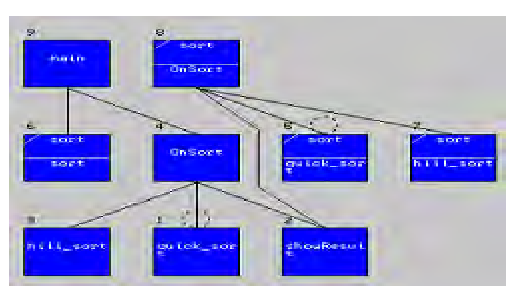


Fig. 21 Bottom-up unit testing order assigned

Appendix 1 provides an example about how to realize 100% of MC/DC (Modified Condition/Decision Coverage) test coverage (we call it J-Coverage here) for a program unit.

- It supports memory leak analysis and memory usage violation check. It is a part of software security testing [9]. An application example is shown in Fig. 22.

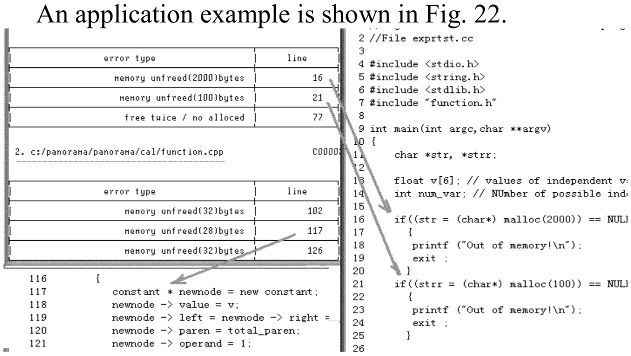


Fig. 22 A report on memory leak and usage violation check

- It supports performance analysis with the capability to report the branch execution frequency to locate performance bottlenecks better as shown in Fig. 23.

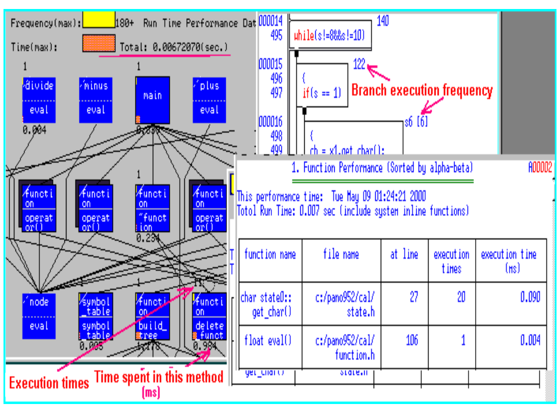


Fig. 23 An application example of performance analysis performed by Panorama++

- 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. 24.

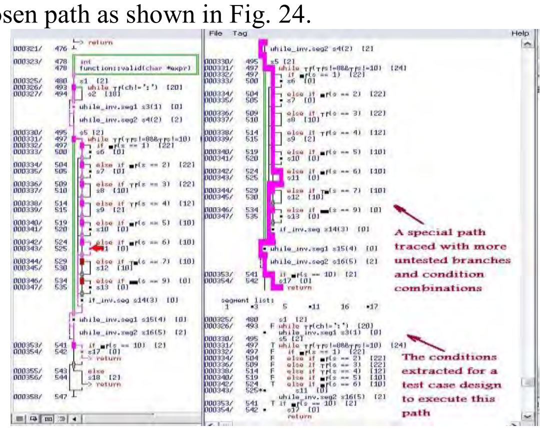


Fig. 24 Assisted test case design performed by Panorama++

- It supports embedded software testing too, as shown in Fig. 25.

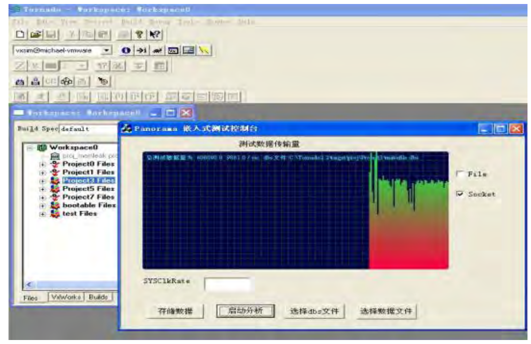


Fig. 25 An application example shows that the MC/DC test coverage data are sent from the target to the test server

- It combines software testing and debugging together visually

The NSE software testing paradigm combines software testing and debugging together closely as shown in the following examples:

(a) The source code of a sample program module "trouble" with seven defects, and the corresponding "main" module is listed as follows:

```
/* File: main.c */

#include <stdio.h>
static char *tp=NULL;
int r=1, x=0, y=1000000, z=0;
FILE *fd=NULL;
void trouble();

main(argc, argv)
int argc;
char **argv;
{
int  k=0;
if(argc>1) trouble(atoi(argv[1]));
if(fd) fclose(fd);
}
```

```
/* File: trouble.c */

/* trouble.c */

#include <stdio.h>
#include <malloc.h>

#ifdef ERROR_SIMULATION
#include "ISA_simu.h"
#endif
extern int x,y,z;
extern FILE *fd;
FILE *fi, *fo;

trouble (x)
int x;
{
int i, t=1;
```

```
char c,*pc=NULL,ch[10],*p=NULL,*e=NULL;
if((e=malloc(4))==NULL)printf("Out of memory,x=%s",x), exit(-1);
for(i = x; i <= 8 && t; p=&ch[i++])
  if(i % 2 ==1) {
     p=&c; t=0; }
ch[0] = *p;  /* seg. fault when x > 8 */
i = x ;
while (i > -2 && i<=7 ){/*dead loop if x=7 or x=3*/
  switch ( x + z ) {
   case  0: case 1: x = z = 1; break;
   case  2: y = 1; break; }
  if ( i < 7 )
    i += 4; }
if ( x < 5 )
  pc = ch;
if( x < 6 )
  fd=fopen("trouble.c", "r");
c = getc (fd);    /* seg. fault when x = 6 */
strcpy (pc, "ab"); /* seg. fault if x = 5 */
c = ch[y]; /* seg. fault when x = 4 */
z = x / z; /* Arith. excep. when x = 2 */
if((p=malloc(3))!=NULL) strcpy(p,"OK");
}
```

(b) The following shows what are provided by a typical test tool using statement / block test coverage metric after the execution that the main() function called the trouble(x) function with x=0 :

```
     #include <stdio.h>
     static char *tp=NULL;
     int r=1, x=0, y=1000000, z=0;
     FILE *fd=NULL;
     void trouble();

     main(argc, argv)
     int argc;
     char **argv;
1 -> {
     int  k=0;
     if(argc>1) trouble(atoi(argv[1]));
1 -> if(fd) fclose(fd);
1 -> }

100.00    Percent of the file executed

     /* trouble.c */
     #include <stdio.h>
     #include <malloc.h>

     #ifdef ERROR_SIMULATION
     #include "ISA_simu.h"
     #endif
     extern int x,y,z;
     extern FILE *fd;
     FILE *fi, *fo;

     trouble (x)
     int x;
1 -> {
     int i, t=1;
     char c,*pc=NULL,ch[10],*p=NULL,*e=NULL;
     if((e=malloc(4))==NULL)printf("Out of memory,x=%s",x), exit(-1);
1,      2 -> for(i = x; i <= 8 && t; p=&ch[i++])
2 ->   if(i % 2 ==1) {
1 ->   p=&c; t=0; }
1 -> ch[0] = *p;    /* seg. fault when x > 8 */
     i = x ;
     while (i > -2 && i<=7 ){/*dead loop if x=7 or x=3*/
2 ->   switch ( x + z ) {
1 ->    case  0: case 1: x = z = 1; break;
1 ->    case  2: y = 1; break; }
2 ->   if ( i < 7 )
2 ->     i += 4; }
1 -> if ( x < 5 )
1 ->   pc = ch;
1 -> if( x < 6 )
1 ->   fd=fopen("trouble.c", "r");
1 -> c = getc (fd);      /* seg. fault when x = 6 */
1 -> strcpy (pc, "ab"); /* seg. fault if x = 5 */
     c = ch[y]; /* seg. fault when x = 4 */
     z = x / z; /* Arith. excep. when x = 2 */
     if((p=malloc(3))!=NULL) strcpy(p,"OK");
1 -> }

100.00    Percent of the file executed
```

It means that the tool offering statement test coverage analysis capability reported 100% of the program have been tested without finding any defect.

(c) Comments on a typical statement / block test coverage analysis tool:

o The analysis result is coding style dependent. Suppose there are two statements as follows:

```
if( 0 ) printf (" Can't be executed. \n");

      and

if( 0 )
      printf ( " Can't be executed. \n");
```

and only the condition parts of them are tested but has never been satisfied, the first statement will report that the entire statement has been tested, but the second one will not.

o It can't identify whether an invisible segment (such as a "if" statement without the "else" part) has been executed or not.

o If several "case" statements share an execution body such as

```
case 0: case 1:
    printf(" Less than 2.\n");
        break;
```

but only one of the conditions of the cases is satisfied (such as case 0: is satisfied), it can't indicate that other cases are not executed.

o It can't identify whether the high end of a loop boundary is executed or not.

o It can't identify whether a condition outcome or the combination of some condition outcomes are executed or not.

(d) After compilation and execution of the program directly with X=6

Without using NSE tools, the system shows an error message with no detailed information (see Fig. 26):

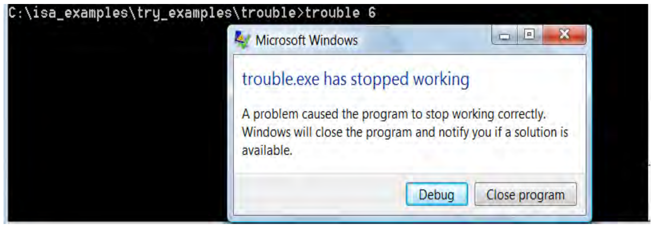


Fig. 26 An error message given by the system without showing the error location

In this case, the system debugger can be used to report the related information in object code format as shown in Fig. 27.

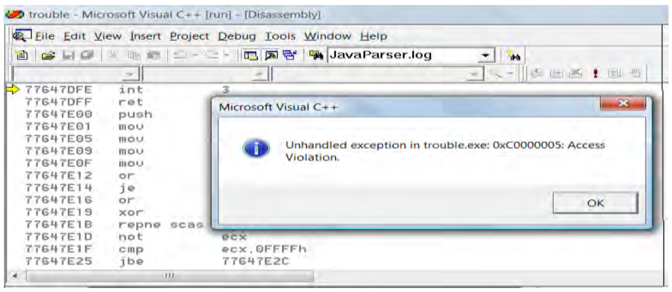


Fig. 27 The system debugger can only show the location of the object code which is not very useful

(e) But with NSE the detailed error information will be reported with the error type and the source code location as shown in Fig. 28

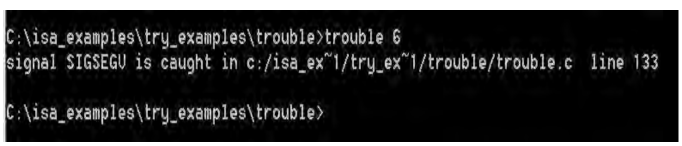


Fig. 28 When it is executed under NSE, an error message is given with the error type and the detailed source code location (line 133 in file trouble.c)

(f) Debugging can also be performed visually with the NSE software engineering paradigm as shown in Fig. 29 to Fig. 33:

See Fig. 29 - after the execution where the main () function called the function trouble(x) with x=0, NSE's support platform Panorama++ will report that only 64% of the program have been tested using the MC/DC test coverage metric.

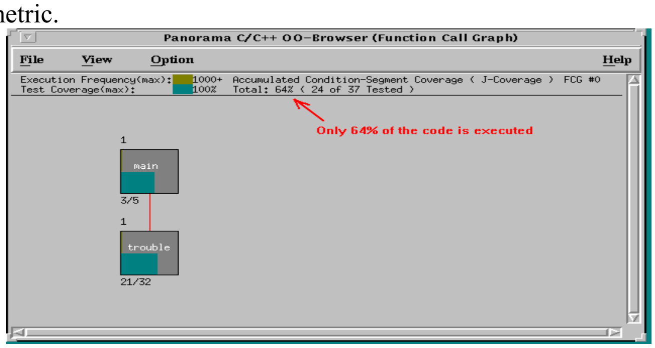


Fig. 29 The corresponding program test coverage shown in J-Chart

See Fig. 30 - the untested branches/segments and conditions can be highlighted in J-diagram.

Fig. 30 The corresponding logic diagram shown in J-Diagram notation with untested branches and conditions highlighted in small black boxes

The untested branches and condition can also be highlighted in a J-Flow diagram as shown in Fig. 31.

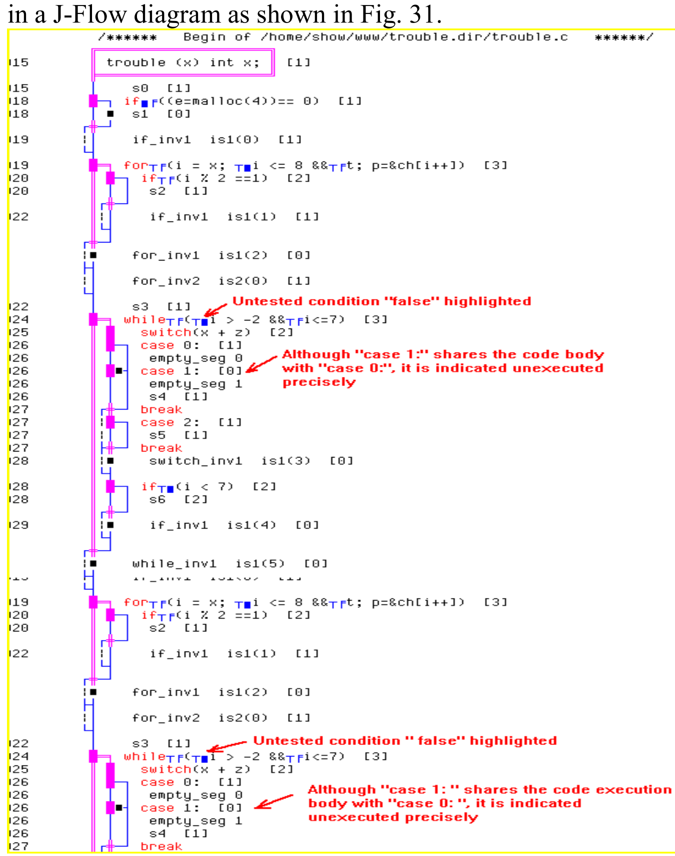


Fig. 31 The corresponding J-Flow diagram shown with the untested branches and conditions highlighted

See Fig, 32 - when a runtime error happens during the testing process, users can directly find the corresponding source code location using J-Flow diagram through searching a word "EXIT" which is automatically added into the J-Flow diagram to indicate the error location (sometimes the defect may be introduced earlier but the program is terminated later).

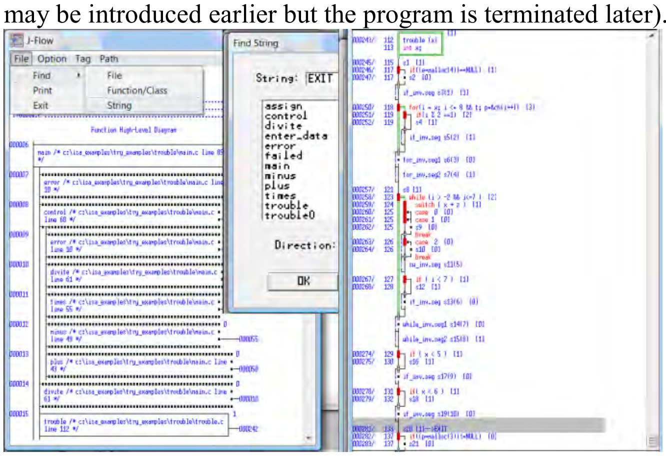


Fig. 32 Finding the location where a program terminated unexpectedly using J-Flow diagram through searching the added word "EXIT"

(g) With all the untested branches and conditions being tested, the seven defects can be found and fixed by modifying the source code. After that the logic diagram will show that 100% of the branches and the conditions are all tested as shown in Fig. 33.

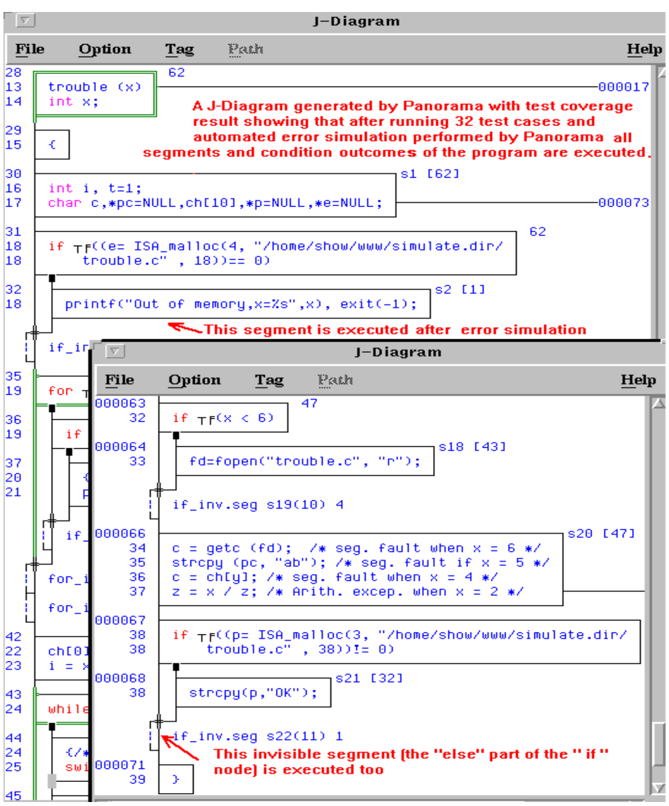


Fig. 33 The final result after removing all defects with the trouble module

V. 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. 34[10].

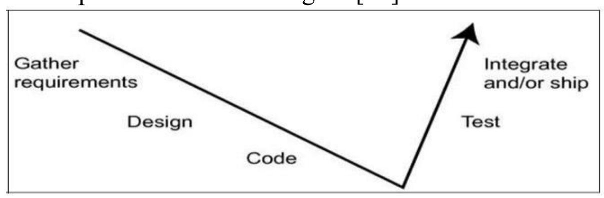


Fig. 34 Traditional software testing performed with incremental software development

The old testing paradigm used for the iterative software development is shown in Fig. 35.

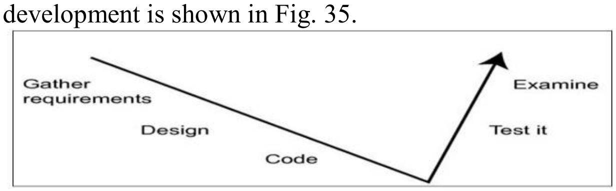


Fig. 35 The old testing paradigm used for the iterative software development[10]

The presented new software testing paradigm used for incremental or iterative software development is shown in Fig. 36.

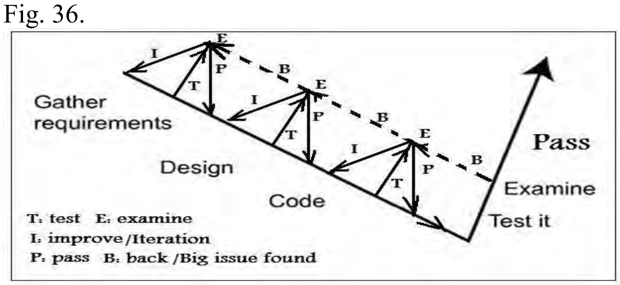


Fig. 36 The presented new software testing paradigm used for incremental or iterative software development

Comparing Fig. 34, Fig. 35, and Fig. 36, 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. 37.

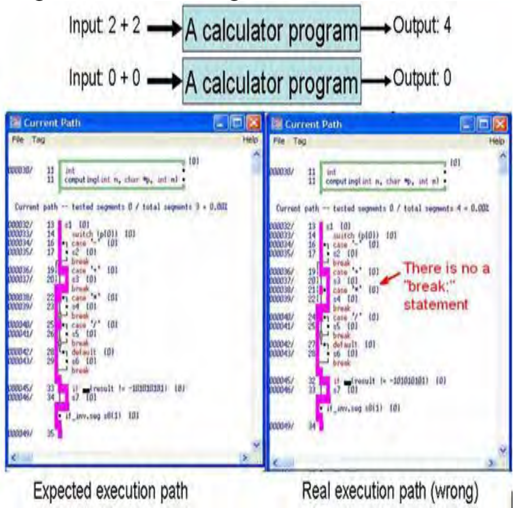


Fig. 37 An application example of transparent-box testing: a bug found even if the output is the same as what is expected (this defect comes from that a "break" statement is missing, so that the result "4" is produced through 2 times 2 rather than 2 plus 2)

(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. 38.

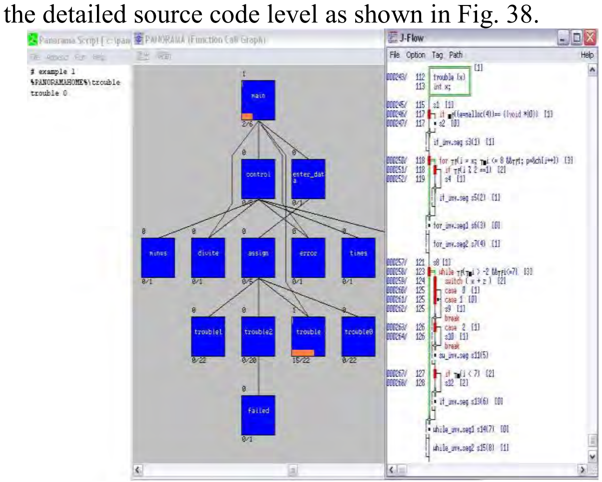


Fig. 38 An example of test coverage analysis result obtained using the presented new software testing paradigm

(the untested branches and conditions are highlighted with small black boxes)

(e) The capability to support automated traceability

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

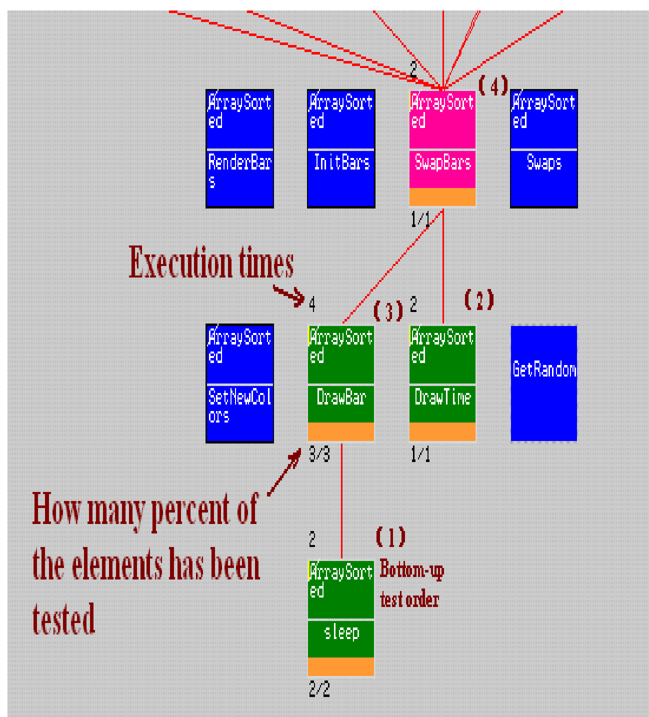
V. Conclusion

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 to find out functional defects, structural defects, and inconsistency defects.

Appendix 1: An example about how to realize 100% of MC/DC (Modified Condition/Decision Coverage) test coverage (we call it J-Coverage here) for a program unit

In this appendix, an example is used for illustrating the test coverage measurement metrics using the NSE support platform Panorama C/C++ for Windows.

With NSE unit testing and integration testing are combined together through Bottom-up unit testing ordering without designing and using stub units:



Here SUM_PRODUCT is a sample program which requests the input of three integers: Low, High and Max. The integers should not be negative, otherwise an error message will be given. When SUM_PRODUCT receives three integers, it outputs for each number k in the

The source code of SUM_PRO.cpp is listed below:

```
#include <stdio.h>
main(void)
// This program prints for each k in the range LOW to HIGH
// k + k and k * k. No more than MAX number of k's are used.
{
 int low, high, max, k, n=0;
 printf("Enter positive integers LOW, HIGH, and MAX:");
 scanf("%d %d %d", &low, &high, &max);
 printf(" LOW = %d  HIGH = %d  MAX = %d \n",low,high,max);
 if ( low >= 0 && high >=0 && max >= 0)
        for (k=low; k<=high; k++)
        {
               ++n;
               if (n > max)
                   break;
               printf("  %d + %d = %d   %d * %d = %d\n",
                k, k, k+k, k, k, k*k);
        }
 else
        printf("Error! The input data are incorrect! \n");
}
```

The Makefile of SUM_PRO.exe is listed below:

```
#"Makefile"
LINK = link32
CC = cl
SUM_PRO.exe: SUM_PRO.cpp
        $(CC) -c SUM_PRO.cpp
        $(LINK) -out: sum_pro.exe -subsystem:console sum_pro.obj libc.lib
kernel32.lib
```

Note: if for Panorama C, the file name SUM_PRO.cpp must be renamed by SUM_PRO.c.

A SUM_PRO.hsi file is generated from the Makefile of SUM_PRO.exe and loaded into the Main Menu of Panorama . Then, a .dbs file is created for SUM_PRO.exe. To capture the dynamic test coverage data, SUM_PRO.exe is executed with several groups of integers as listed below:

LOW	HIGH	MAX
2	8	0
10	20	12
10	1	11
2	8	-2
2	-2	8
-2	2	8

A series of J-Flow and J-Diagrams in OO-Diagrammer are listed to show the changes of accumulated test coverage each time when SUM_PRO.exe is executed.

Note: In this Appendix, the test coverage refers to the Accumulated test coverage in order to show the result of all the executions.

Before the execution of SUM_PRO.exe, the test coverage of the code is zero. This is reflected in the bar graph and diagrams below:

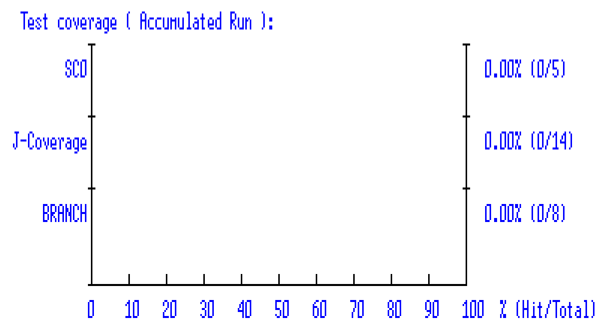


Figure A-1. Bar graph in OO-Diagrammer:
The test coverage data are all zero.

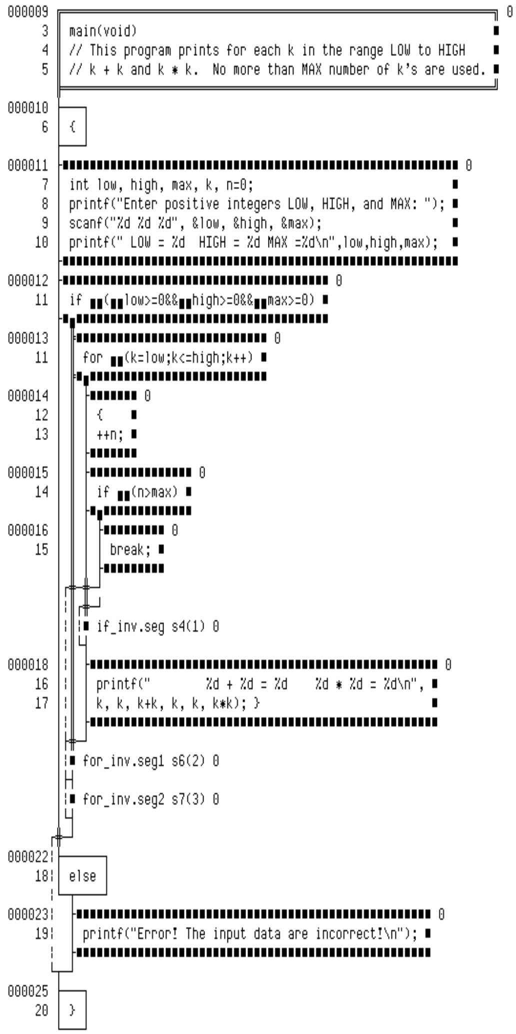


Figure A-2. J-Diagram in OO-Diagrammer:
Accumulated test coverage: All the elements are untested and highlighted.
To execute the sample program, type SUM_PRO.exe under appropriate directory at prompt:
C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: **2 8 0**

LOW = 2 HIGH = 8 MAX =0

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.

Then check the Bar graph, J-Flow and J-Diagram in OO-Diagrammer. Select the Accumulated test coverage on the corresponding Options dialog box, then click OK. The test coverage data are automatically updated:

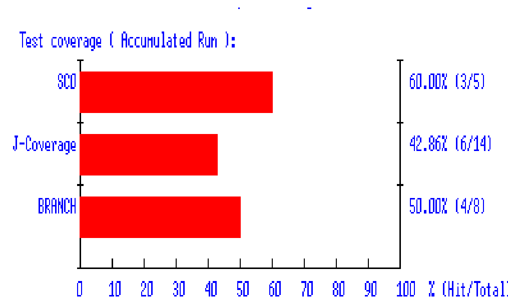


Figure A-3. Bar graph in OO-Diagrammer:
After the first execution of sum_pro.exe, the test coverage results are to be improved.

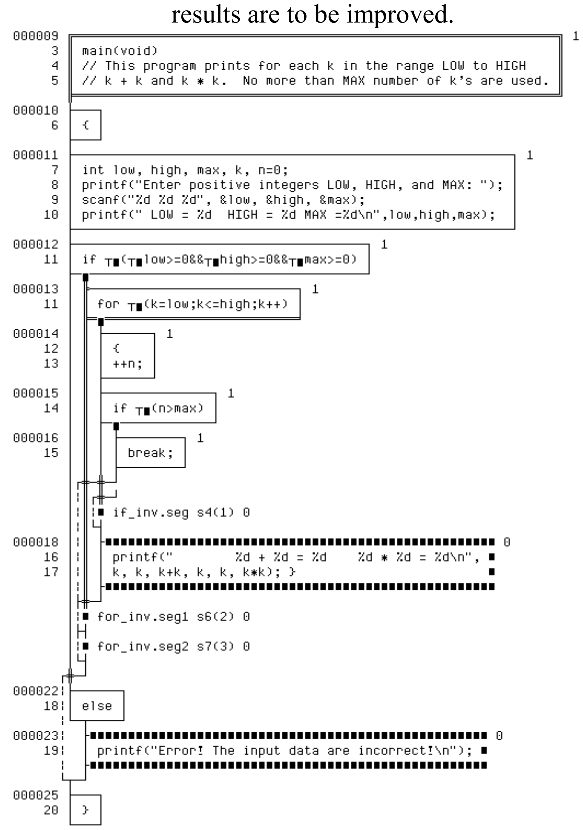


Figure A-4. J-Diagram in OO-Diagrammer:
After the first execution of sum_pro.exe.

Now, execute SUM_PRO.exe again. This time three integers 10, 20, and 12 are inputted. SUM_PRO.exe outputs, from 10 to 20, 11 groups of equations:

C: >\Func\SUM_PRO\sum_pro.exe

*Enter positive integers LOW, HIGH, and MAX:***10 20 12**

LOW = 10 HIGH = 20 MAX =12
*10 + 10 = 20 10 * 10 = 100*
*11 + 11 = 22 11 * 11 = 121*
*12 + 12 = 24 12 * 12 = 144*
*13 + 13 = 26 13 * 13 = 169*
*14 + 14 = 28 14 * 14 = 196*
*15 + 15 = 30 15 * 15 = 225*
*16 + 16 = 32 16 * 16 = 256*
*17 + 17 = 34 17 * 17 = 289*

*18 + 18 = 36 18 * 18 = 324*
*19 + 19 = 38 19 * 19 = 361*
*20 + 20 = 40 20 * 20 = 400*

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Then check the Bar graph, J-Flow and J-Diagram in OO-Diagrammer. Select the Accumulated Test Coverage Data on the corresponding Options dialog box, then click OK. The test coverage data on the diagrams are automatically updated:

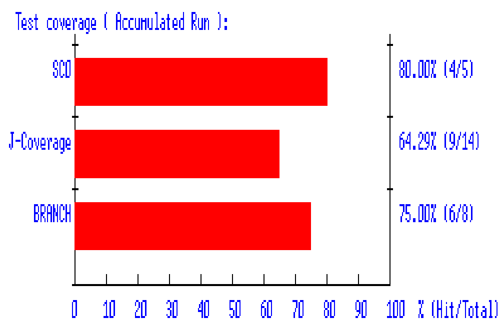


Figure A-5. Bar graph in OO-Diagrammer:
The test coverage data have increased significantly.

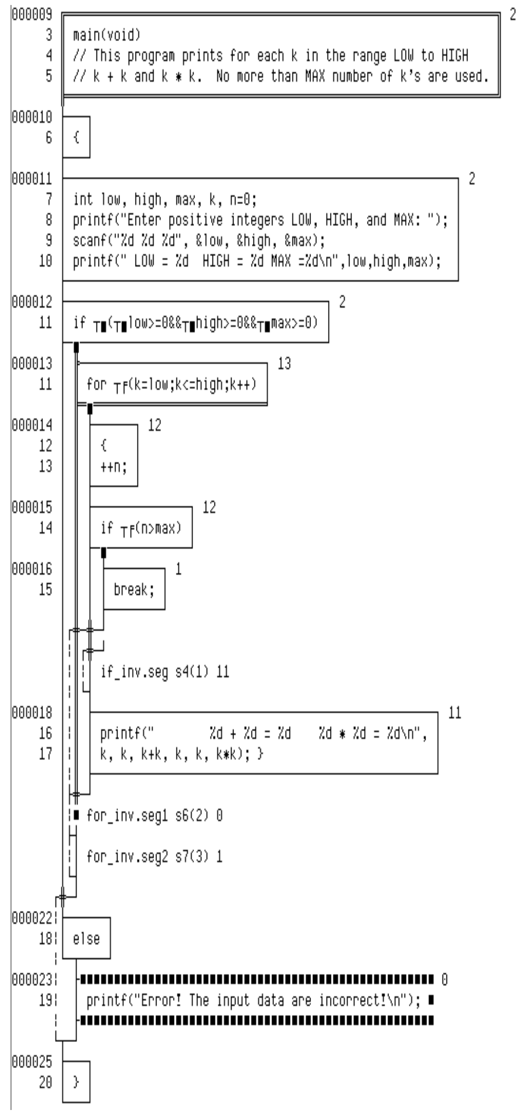


Figure A-6. J-Diagram in OO-Diagrammer:

Accumulated Test Coverage: The number of unexecuted elements highlighted has been greatly decresed compared to the diagrams before.

Now, execute SUM_PRO.exe again to increase its test coverage furthermore. This time integers 10, 1, 11 are inputted.

C: >\Func\SUM_PRO\sum_pro.exe

*Enter positive integers LOW, HIGH, and MAX:***10 1 11**

LOW = 10 HIGH = 1 MAX =11

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since Low=10 > High=1, no equation is outputted this time.

Then check the Bar graph, J-Flow and J-Diagram in OO-Diagrammer. Select the Accumulated Test Coverage on the corresponding Options dialog box, then click OK. The test coverage data are automatically updated:

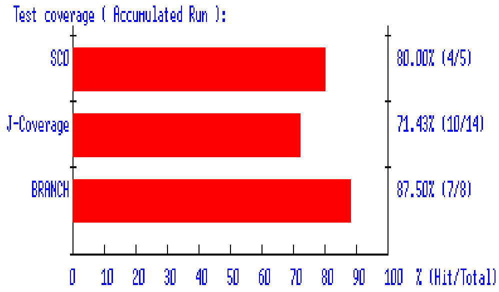


Figure A-7. Bar graph in OO-Diagrammer:

Accumulated Test Coverage: Compared to Figure A-6, one more branch and one more segment are tested. Consequently, J-Coverage is increased by one too.

Accumulated test coverage: Compared to Figure A-6, one more segment (branch) is tested.

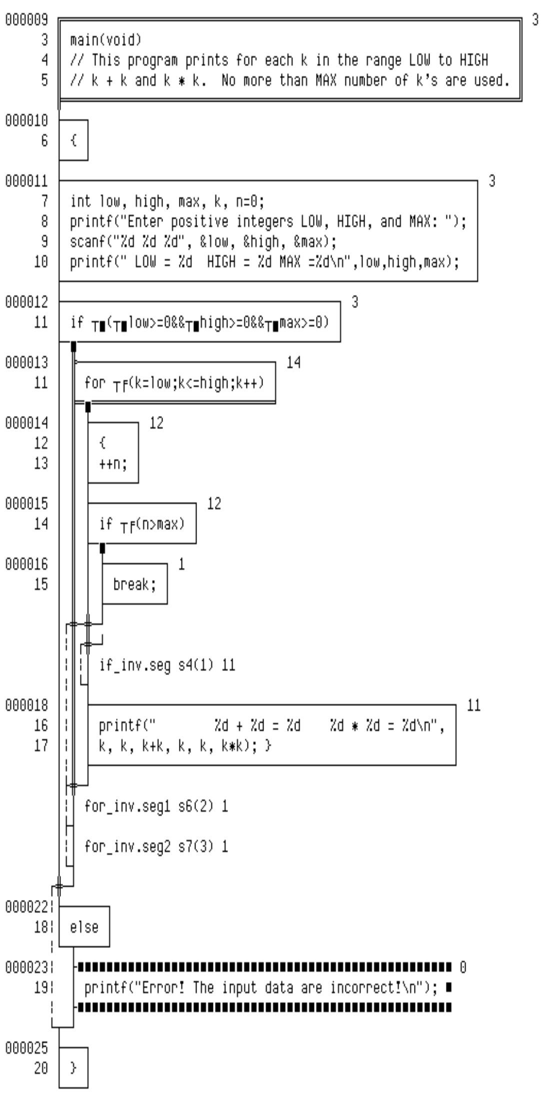


Figure A-8. J-Diagram in OO-Diagrammer:

Accumulated Test Coverage: Compared to Figure A-6, one more segment (branch) is tested.

Now, carefully observe the J-Flow or J-Diagram, you may find out that the condition test coverage should be increased. Since Condition True has reached 100% coverage, the Condition False needs to be increased.

C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: **2 8 -2**

LOW = 2 HIGH = 8 MAX =-2

Error! The input data are incorrect!

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since a negative integer is inputted, an error message is given this time.

Then check the Bar graph, J-Flow and J-Diagram in OO-Diagrammer. Select the Accumulated test coverage on the corresponding Options dialog box, then click OK. The test coverage data are automatically updated:

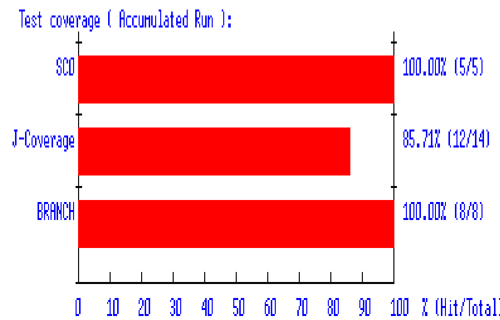


Figure A-9 Bar graph in OO-Diagrammer:

The accumulated test coverage of SC0, branch have reached 100%. J-Coverage is increased too.

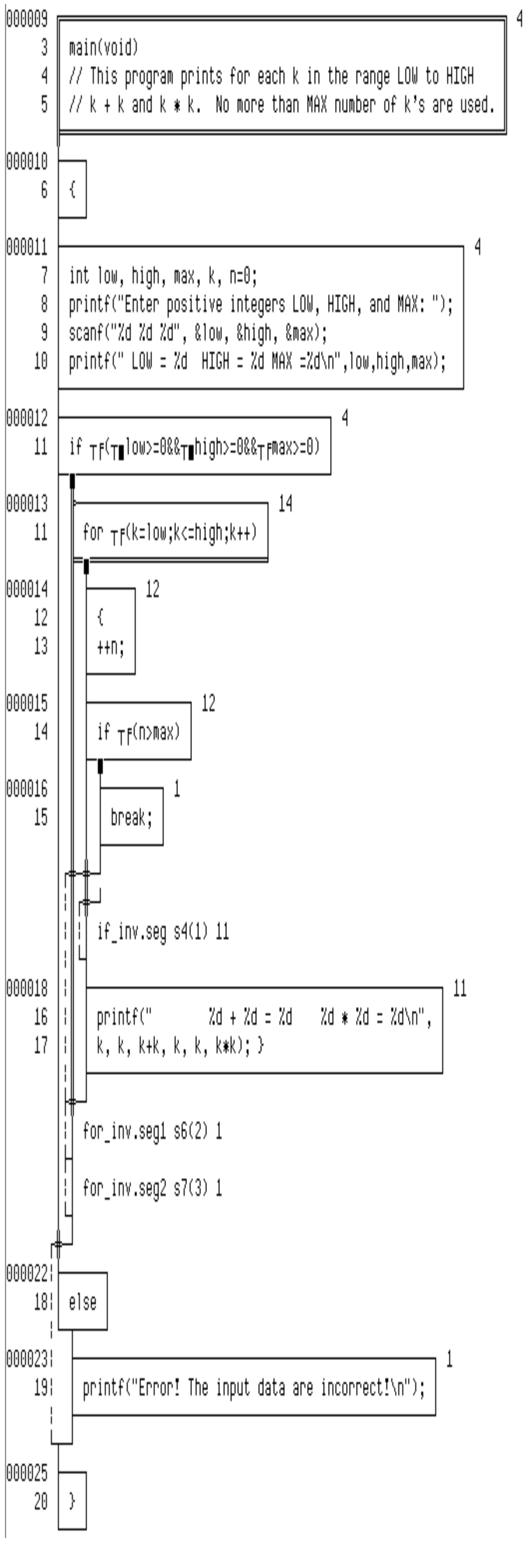


Figure A-10. J-Diagram in OO-Diagrammer:
Accumulated Test Coverage: only 2 conditions are untested.

To increase the coverage of Condition False, run SUM_PRO.exe again and input another group of integers. This time, integer High is negative.

C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: **2 -2 8**

LOW = 2 HIGH = -2 MAX =8

Error! The input data are incorrect!

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since negative integer High is inputted, an error message is given too.

Then check the Bar graph, J-Flow and J-Diagram in OO-Diagrammer. Select the Accumulated test coverage in the corresponding Options dialog box, then click OK. The test coverage data are automatically updated:

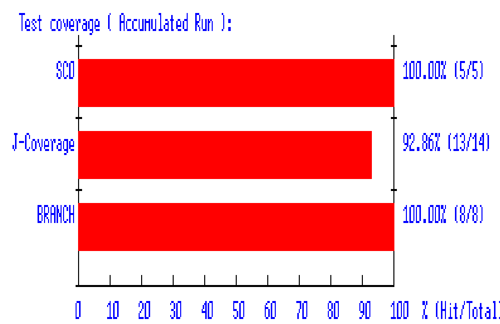


Figure A-11. Bar graph in OO-Diagrammer:

J-Coverage has been increased.

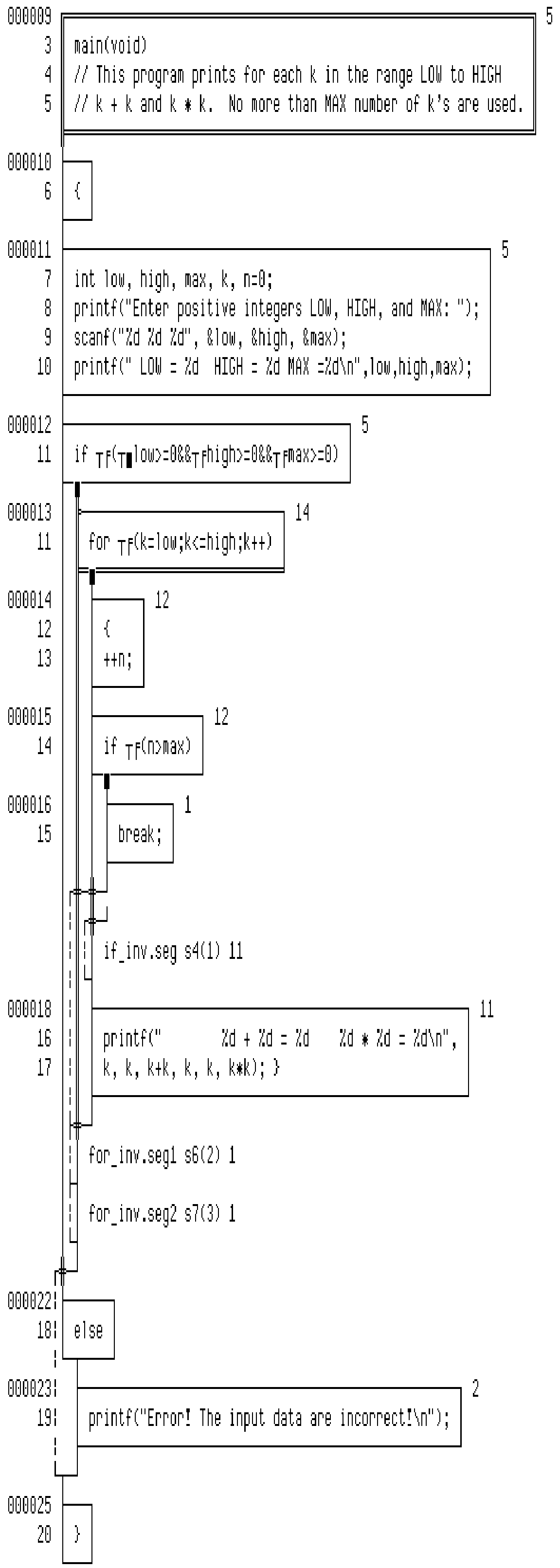


Figure A-12. J-Diagram in OO-Diagrammer:
Accumulated Test Coverage: only 1 False condition is untested.

To cover all the conditions, run SUM_PRO.exe again and input another group of data with negative Low integer.

C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: **-2 2 8**

LOW = -2 HIGH = 2 MAX =8

Error! The input data are incorrect!

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since negative integer Low is inputted, an error message is given too.

Then check the Bar graph, J-Flow and J-Diagram in OO-Diagrammer. Select the Accumulated test coverage on the corresponding Options dialog box, then click OK. All the conditions should have been covered:

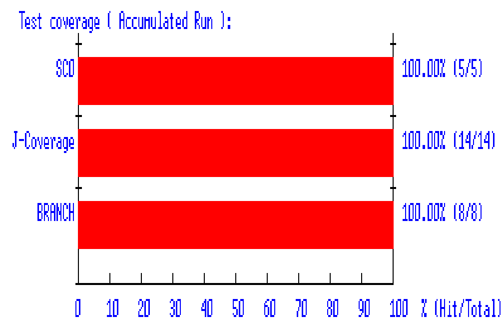


Figure A-13. Bar graph in OO-Diagrammer:

Accumulated test coverage: all the test coverage metrics have been reached 100%.

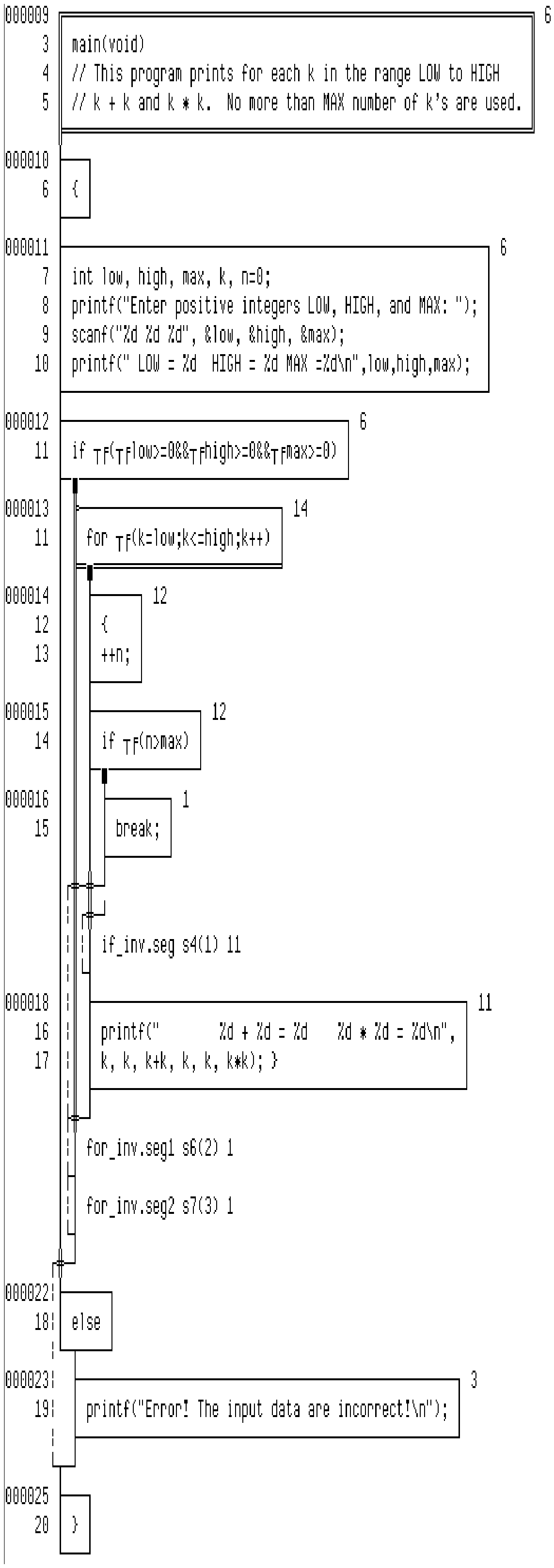


Figure A-14. J-Diagram in OO-Diagrammer:
Accumulated test coverage: The program sum_pro.exe is completely tested.

From the example above, it is clearly shown how test coverage data are displayed on J-Flows and J-Diagrams, and how the result shown may help you to increase the coverage of your program.

Similarly, other tools of Panorama C/C++, such as the structure charts, software metrics diagrams, reports, ActionPlus diagrams, etc., can also show the dynamic test data vividly and help you successfully plan the further testing.

REFERENCES

[1] RUSLI ABDULLAH (2012), Towards Developing Software Testing As a Service (Staas) Model in Cloud Computing: A Case of Collaborative Knowledge Management System.

[2] Glenford J. Myers ,(2004) The Art of Software Testing, Second Edition, John Wiley & Sons, Inc.

[3] Jay Xiong, Tutorial, *A Complete Revolution in Software Engineering Based on Complexity Science,* WORLDCOMP'09, Las Vegas, July 13-17, 2009.

[4] JOAN F. CHIPIA LOBO, GLORIA MOUSALLI-KAYAT, and FRANCKLIN RIVAS (2011), Methodologies development and software quality metrics in educational applications.

[5] Palmer, J.D. "Traceability." *Software Requirements Engineering*. Richard H. Thayer and Merlin Dorfman, eds. New York: IEEE Computer Society Press, 1997.

[6] Andrew Kannenberg, Garmin International Dr. Hossein Saiedian, The University of Kansas, *Why Software Requirements Traceability Remains a Challenge,* CrossTalk, Jul/Aug 2009 Issue.

[7] RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification, RTCA, Washington D.C., USA, 1992.

[8] Yuhei Otani, Hiroaki Hashiura, and Komiya Seiichi (2011), A Software Testing Tool with the Facilities to Restore the State at Program Execution of a Program under Test. [9] DONGHONG LI 1, SHUANGHUI YI (2011), Security Functional Testing of Distributed software,.

[9] Alistair Cockburn, *Using Both Incremental and Iterative Development,* CrossTalk, May 2008

Jay Xiong , the President of NSEsoftware, LLC, USA and the President of Aisai Shanghai, Ltd. He has brought his 20 years of experience in CAD/EDA to software engineering automation with his innovative techniques for nonlinear software engineering, graphical representation, software testing, quality assurance, and maintenance. Trained at Zhong Shan University and in integrated circuit design at the Chinese Academy of Science, Jay Xiong invented the "Shortest Path Routing Algorithm Using Wave Diffraction" at the Hitachi Research Center in Japan. This major technical achievement brought him to the University of California, Berkeley as the foremost Chinese scientist in the Computer Aided Integrated Circuit Layout Project jointly sponsored by The National Science Foundation of the United States and the Chinese Academy of Science. He founded Advanced Software Automation, Inc. (1987) and International Software Automation, Inc. (1992) in Silicon Valley. He is the designer of Hindsight and Panorama products ("Panorama : developed by International Software Automation, Inc. encompasses a complete set of tools for object-oriented software development including tools that assists test case design and test planning." (ROGER S. PRESSMAN, "Software Engineering: A Practitioner's Approach"). Being invited, he offered a tutorial to WORLDCOMP'09 with the title "**Complete Revolution in Software Engineering Based on Complexity Science**" (http://www.world-academy-of-science.org/worldcomp09/ws/tutorials/tutorial_xiong).He is the author of the book, "New Software Engineering Paradigm Based on Complexity Science", published in 2011 by Springer in US (http://www.springer.com/physics/complexity/book/978-1-4419-7325-2).

Lin Li , the CEO of Aisai Shanghai Ltd, China, She is an inventor of several inventions. She is the co-author of the following papers published in the 1st International Conference on Innovative Computing and Information Processing (INCIP '13) to be held at Rhodes Island, Greece, July 16-19, 2013 (http://naun.org/wseas/cms.action?id=4593):

(1) Nonlinear and Quantitative Software Engineering Method Based on Complexity Science

(2) Automated Generation of Software Documents Consistent with and Traceable to and from Source Code

(3) Transparent-Box Method Combining Structural and Functional Software Testing together Seamlessly

: