Method and practice on safety software verification & validation for digital reactor protection system

LI Duo, ZHANG Liangju, FENG Junting

Institute of Nuclear and New Energy Technology, Tsinghua University., Beijing 100084, China
E-mail:{ liduo, zhanglj, fengjt}@tsinghua.edu.cn

Abstract: The key issue arising from digitalization of reactor protection system for Nuclear Power Plant (NPP) is in essence, how to carry out Verification and Validation (V&V), to demonstrate and confirm the software is reliable enough to perform reactor safety functions. Among others the most important activity of software V&V process is unit testing. This paper discusses the basic concepts on safety software V&V and the appropriate technique for software unit testing, focusing on such aspects as how to ensure test completeness, how to establish test platform, how to develop test cases and how to carry out unit testing. The technique discussed herein was successfully used in the work of unit testing on safety software of a digital reactor protection system.
Keywords: Software Reliability, Verification, Digital Instruments

1 - Introduction

As the instrumentation and control system (I&C) of nuclear power plant (NPP) comes into a digital era, the key issues arising from digitalization of NPP I&C are how to effectively demonstrate and confirm the completeness and correctness of the software that performs reactor safety functions. In addition investigation of which characteristic safety software should have to be approved by the nuclear safety authority is important. It is commonly accepted that the essential way to address these issues is to carry out a strict and independent Verification and Validation (V&V) process in parallel with software development phases. Among other approaches the most important activity of software V&V process is software unit testing, which verifies the consistency, correctness and completeness of the software coding with software design specification, yielded from the preceding stages of software development process.

The practical work on V&V for the software of a digitalized reactor protection system is covered in this paper. The first part of the paper outlines the V&V plan, including V&V processes, activities and tasks that are carried out in parallel with the safety software development. The second part focuses on software unit testing, showing the technique of how to ensure test completeness, how to establish test platform, how to develop test cases and how to carry out unit testing. The last part demonstrates the actual test practice with a typical unit testing.

2 - Safety software V&V

Software V&V is a disciplined approach to assess software products throughout the product life cycle. The V&V processes are tailored to specific system requirements and applications and carried out in parallel with software development phases. The V&V activities for each process are carefully designed to certify that the objectives of each development phase are implemented correctly. Software V&V processes, activities and tasks are defined in IEEE standard and guidelines [1][2].

The development of safety software for NPP should be a process controlled every step of the way. The development process is organized as an ordered collection of distinct phases. Each phase uses the information developed in the earlier phase and provides input information for preceding phases. Typical phases of the development steps and an outline of the process that may be applied are shown in Fig. 1. The boxes show the development activities to be performed and the hollow arrows show the intended order and the primary information flow. A V&V effort is typically applied in parallel with software development and support activities, therefore the boxes show the related V&V activities as a whole

Received date :October 15, 2009
(Revised date : November 3, 2009)

and the intended order of these activities, and their potential information flow are shown by the solid arrows. Software V&V activities determine whether development products of a given activity confirm to the requirements of that activity, and whether the software satisfies its intended use and user needs. This determination may include analysis, evaluation, review, inspection, assessment and testing of software products and processes. V&V processes assess the software in the context of the system, including the operational environment, hardware, interfacing software, operators, and users.

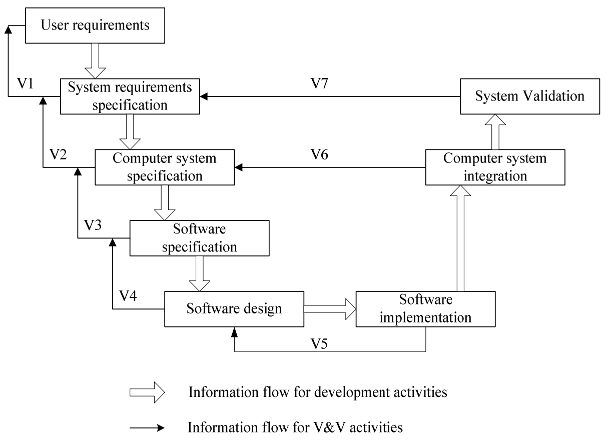

User requirements
System requirements specification
Computer system specification
Software specification
Software design
Software implementation
Computer system integration
System Validation
V1
V2
V3
V4
V5
V6
V7
Information flow for development activities
Information flow for V&V activities

Fig.1 Software development processes and its V&V activities

The objective of V&V activities is to collect evidence and prepare documentations to be used in the safety demonstration for the software for all various phases of the systems life cycles. The general document for a V&V effort is the Software V&V Plan (SVVP), which includes all general information for a V&V effort (e.g., purpose, references, definitions, organization, schedule, and etc.), identifies V&V activities and tasks to be performed for all software life cycle processes, and contains all V&V documentation to be produced, such as V&V reports and V&V test documentation.

3 - Safety software unit testing

One of the important V&V processes is the V&V for software implementation or software coding V&V, which is coded as V5 in Fig.1. The development activities in software implementation phase translate software design into software source code, and finally release that information as a piece of code suitable for a machine. The V&V activities in parallel with software implementation phase verify the consistency and correctness of above mentioned translations, making sure the functions of each software model defined in software design were implemented correctly, and no errors were introduced during software coding, required programming practice, prudent checking and criterion were fully carried out.

The main objects of software implementation V&V include tracing source code to verify its consistency with software design, and detecting errors introduced during software coding, which are mainly performed through static analysis and dynamic testing of software units. Software unit testing should be carried out on each basic code units one by one and cover implementation details, such as logical structure and data flow inside a software unit, which is kind of a white box testing, and is different from black box testing that focuses only on inputs and outputs of a software unit.

The essential demand for unit testing of safety software is the completeness of the testing and is measured in coverage ratio, which should be 100% for safety software. However, there are many different kinds of coverage, each focusing on one aspect of software source code. One kind of coverage can not be included or replaced by another. The strategy to determine what kinds of coverage ratio should be included for the completeness of the test is based on the analysis of specific features for a software source code.

For the sake of easy realization of complete testing, we use modular and simple structure in the development of the safety software:

(1) The software was divided into several basic modules.

(2) Only straight forward top-down flow of execution was used in each module, no nesting, no jumping, no interrupt was used, making the process of execution and its behavior determinable and transparent.

(3) No operation system was used; no system routines were called for the execution of the safety software, so that all source codes are available and testable.

Based on these characteristics of the software, it is possible to carry out a complete unit testing with 100% coverage ratio, including statement coverage, branch coverage and MC/DC coverage.

Statement coverage means that test cases are designed to let every executable code line execute once at least, and all executable code lines would be tested when test cases make statement coverage ratio achieve 100%. However, all executable code lines do not execute one after the other, since there are some code branches in the software. So statement coverage cannot be used to measure whether a code branch executing correctly a branch coverage should be added to measure the completeness of safety software unit testing.

Branch coverage, or decision coverage, means that test cases are designed to make true value and false value of every decision conditions can be achieved, i.e. code branches execute with true value and false value once at least, and all code branches would be tested when test cases make branch coverage ratio achieve 100%. However, when a code branch is controlled by a composite decision condition and the execution of branch with true value or false value is decided by two or more condition composite values, branch coverage could be used to measure only one possible composite condition value rather than all condition values.

Decision condition coverage is used to measure and test the composite decision condition, which means that test cases are designed to make each condition value in a composite decision condition taking all values, whether true or false value once at least, and the result of composite decision condition itself, taking true and false value once at least. In theory, decision condition coverage can be used to measure whether a composite decision condition is completely tested, however, test cases would have an exponential increase with the number of decision condition, which makes it is difficult to achieve 100% decision condition coverage ratio. In safety software unit testing we instead choose modified conditions/decision coverage.

Modified condition/decision coverage, or MC/DC, is developed from decision condition coverage with such changes, that test cases are designed to make the entry and exit of every code branch executed once at least, every decision result is executed once at least, and make basic Boolean expression, rather than each condition itself taking true and false value once at least. Basic Boolean expressions are decomposed from a composite decision condition through logical analysis, and each expression has an independent value for the condition result of the composite decision, which prevents test cases from exponentially increasing and makes it possible to achieve 100% MC/DC coverage ratio.

4 Establish software unit testing environment

In general, a software unit to be tested is not an independently executable program, so it is necessary to establish a testing environment to make the software unit executable. This includes the development of driver modules and stub modules. In a testing environment a driver module works as a main program getting input data from test cases, passing the data to the software unit and outputting test results, and stub modules work as replacements of subroutines to be called by the software unit, which is shown in Fig. 2. There are three steps to establish a testing environment, which are developing a driver module to establish a minimal executable system and simulate a up-top subroutine, calling the software unit that needs to be tested, developing stub modules to simulate the interface of the software unit and subroutines to be called by the software unit, and preparing input data, or test cases, to make the software unit executable in the testing environment.

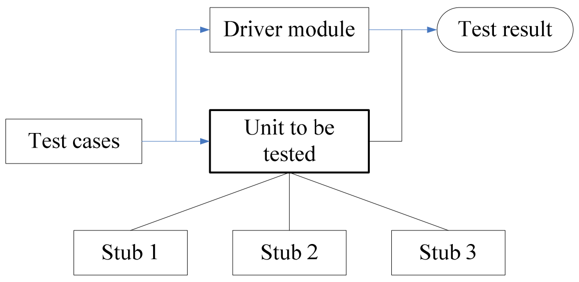


Fig.2 Unit testing scheme

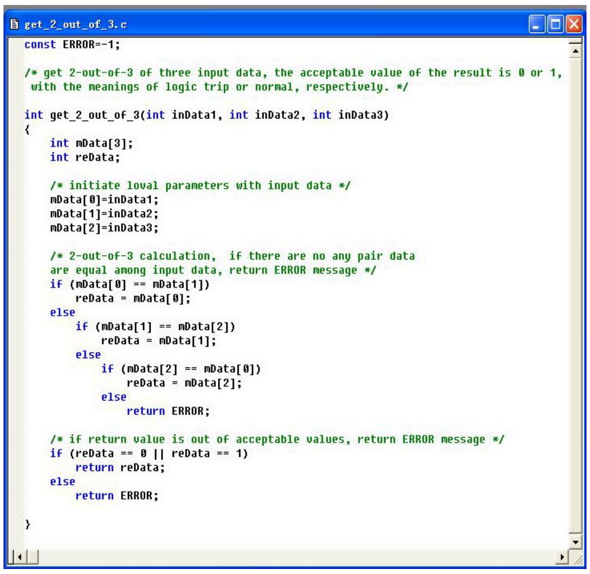
The work of safety software unit testing mainly include the steps of designing test cases, editing input and anticipated output values for the test cases to check whether the software unit executes correctly, analyzing what test cases are needed to achieve the testing goal for a safety software, i.e. 100% statement coverage, branch coverage and MC/DC coverage ratios in the testing result. Test cases should be designed for each safety software unit through following steps:

(1) According to the software unit design document, make clear the function, value range of input and anticipated output data of the software unit.

(2) Design the first test case and let the software unit execute with the input data within its designed value range and validate the function of the software unit being achieved.
(3) Analyze the test result according to software source code and flow diagram, and design a new test case by changing input data to make a certain statement, branch, or composite decision condition in a branch execute and increase related coverage ratio.
(4) Each test case is designed to increase a kind of coverage ratio; the result of the testing should be checked for the coverage ratio. If a test case has no contribution to increase any coverage ratio, it should be modified or replaced by another one.
(5) Repeat above steps (3) to (4) till 100% statement coverage, branch coverage and MC/DC coverage ratios are achieved.

5 - Practice of software unit testing

We choose VectorCAST as an aiding tool to perform software unit testing in the V&V of the safety software of a digital reactor protection system. VectorCAST is a Commercial Off-the-Shelf(COTS) software tool that provides an integrated software test solution. VectorCAST has a summary report to visualize what and how the test coverage is achieved, in addition to some useful utilities, such as developing test driver and stub automatically.
The procedures of software unit testing for an example program are shown in the following.
The example program "get_2_out_of_3.c" is a function routine to execute 2_out_of_3 logical process in a digital reactor protection system. The source code and flow diagram are shown in Figs. 3 and 4, respectively. In the program there are 3 processes from top to down:
(1) Initiate 3 local parameters mData[0,1,2] according to input data, whose available values are 0 or 1 ;
(2) Compare every 2 parameters value among mData[0,1,2], if there are any two parameters having equal value, put the value into the parameter of reData, which is going to be returned by the program, otherwise put the "-1", e.g. ERROR, into the reData;
(3) Validate the value of reData is acceptable, i.e. it is 0 or 1 and return reData, otherwise return -1, e.g. ERROR.
Based on the aiding tool of VectorCAST we establish a testing environment for the program of "get_2_out_of_3.c" and carry out the unit testing in 4 steps:
(1) Set compile option with the same parameters used in the final compile of "get_2_out_of_3.c";
(2) There are no stub routines since "get_2_out_of_3.c" calling nothing;
(3) VectorCAST automatically builds a driver module to link with "get_2_out_of_3.c", which is going to be executed with a set of test cases;
(4) Design and edit test cases executed by VectorCAST, record testing result. The test cases are developed manually and VectorCAST gives some suggestions or hints to develop new test cases based on coverage analysis for the previous tests.



```
const ERROR=-1;

/* get 2-out-of-3 of three input data, the acceptable value of the result is 0 or 1,
 with the meanings of logic trip or normal, respectively. */

int get_2_out_of_3(int inData1, int inData2, int inData3)
{
    int mData[3];
    int reData;

    /* initiate loval parameters with input data */
    mData[0]=inData1;
    mData[1]=inData2;
    mData[2]=inData3;

    /* 2-out-of-3 calculation,  if there are no any pair data
    are equal among input data, return ERROR message */
    if (mData[0] == mData[1])
        reData = mData[0];
    else
        if (mData[1] == mData[2])
            reData = mData[1];
        else
            if (mData[2] == mData[0])
                reData = mData[2];
            else
                return ERROR;

    /* if return value is out of acceptable values, return ERROR message */
    if (reData == 0 || reData == 1)
        return reData;
    else
        return ERROR;

}
```

Fig.3 Source code of example program

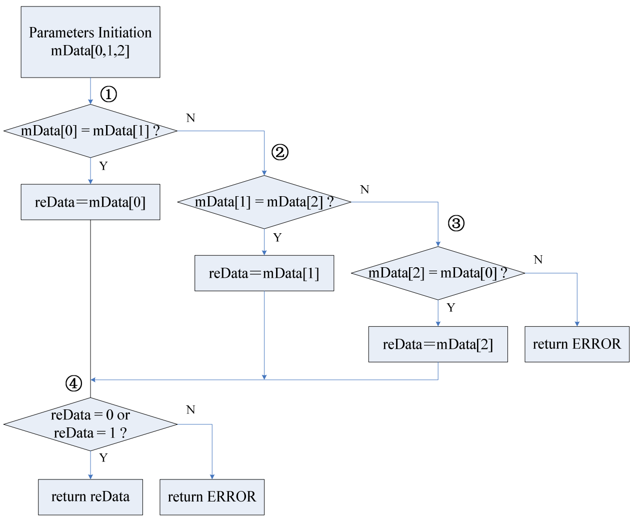
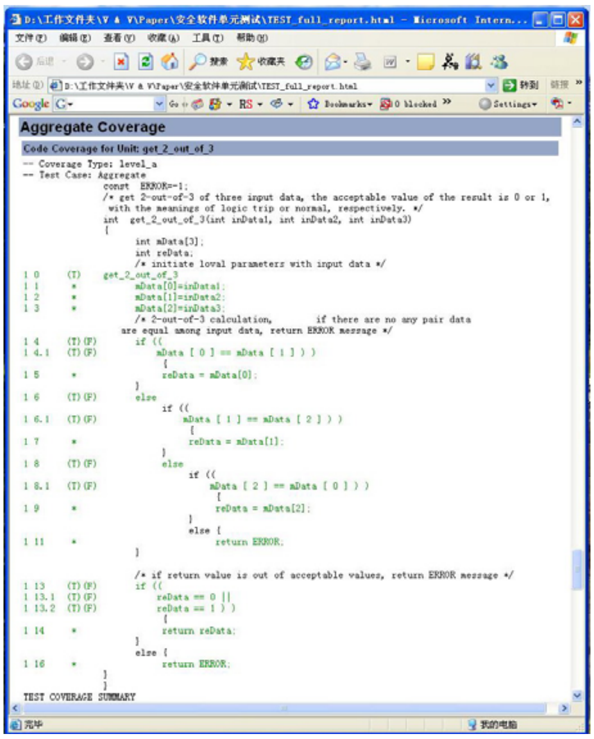


Fig.4 Flow diagram of example program

Table 1 Testing case design for example program

	Test cases				Braches to be tested					
	Data[0]	Data[1]	Data[2]	return	①	②	③	④	reData==0	reData==1
1.	0	1	2	-1	F	F	F	-	-	-
2.	1	0	1	1	F	F	T	T	F	T
3.	2	1	2	-1	F	F	T	F	F	F
4.	0	1	1	1	F	T	-	T	F	T
5.	0	0	1	0	T	-	-	T	T	F

As shown in Fig.4, there are 4 code branches in the source code of the example program, and there is a composite decision condition in branch ④, therefore, the test cases should be designed to make each code branch taking true and false value at least once, and make each decision condition, i.e. reData = = 0 and reData = = 1, taking true and false value once. The 5 test cases are designed and shown in Table 1, and the tested code branches and decision conditions for each test case are shown, too. Testing result is shown in Fig. 5.


```
Aggregate Coverage
Code Coverage for Unit: get_2_out_of_3
-- Coverage Type: level_a
-- Test Case: Aggregate
                 const  ERROR=-1;
                 /* get 2-out-of-3 of three input data, the acceptable value of the result is 0 or 1,
                  with the meanings of logic trip or normal, respectively. */
                 int  get_2_out_of_3(int inData1, int inData2, int inData3)
                 {
                      int mData[3];
                      int reData;
                      /* initiate loval parameters with input data */
1 0      (T)     get_2_out_of_3
1 1       *           mData[0]=inData1;
1 2       *           mData[1]=inData2;
1 3       *           mData[2]=inData3;
                      /* 2-out-of-3 calculation,       if there are no any pair data
                    are equal among input data, return ERROR message */
1 4      (T)(F)       if ((
1 4.1    (T)(F)           mData [ 0 ] == mData [ 1 ] ) )
                           {
1 5       *                reData = mData[0];
                      }
1 6      (T)(F)       else
                           if ((
1 6.1    (T)(F)                mData [ 1 ] == mData [ 2 ] ) )
                                {
1 7       *                     reData = mData[1];
                           }
1 8      (T)(F)            else
                                if ((
1 8.1    (T)(F)                    mData [ 2 ] == mData [ 0 ] ) )
                                     {
1 9       *                          reData = mData[2];
                                }
                                else {
1 11      *                          return ERROR;
                      }

                      /* if return value is out of acceptable values, return ERROR message */
1 13     (T)(F)       if ((
1 13.1   (T)(F)           reData == 0 ||
1 13.2   (T)(F)           reData == 1 ) )
                           {
1 14      *                return reData;
                      }
                      else {
1 16      *                return ERROR;
                 }
                 }
TEST COVERAGE SUMMARY
```
D:\工作文件夹\V & V\Paper\安全软件单元测试\TEST_full_report.html - Microsoft Intern...
文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)
后退
搜索
收藏夹
地址(D) D:\工作文件夹\V & V\Paper\安全软件单元测试\TEST_full_report.html
转到
链接
Bookmarks
0 blocked
Settings
完毕
我的电脑

Fig.5 Coverage analysis for testing result of example program

where asterisks indicate that lines are covered; (T) and (F) indicate that branches with True and False values are covered respectively.

6 - Conclusion

Strict and independent V&V processes are necessary to effectively demonstrate and confirm the completeness and correctness of the software that performs reactor safety functions. Among others the most important activity of software V&V is, software unit testing, which verifies the consistency, correctness and completeness of the software coding with software design specification yielded from the preceding stage of software development process. This paper discusses the method for software unit testing, which was successfully demonstrated and used in the work unit of testing, on safety software of a digital reactor protection system.

References

[1] IEEE Std 1012-1998, IEEE Standard for Software Verification and Validation

[2] IEEE Std 1059-1993, IEEE Guide for Software Verification and Validation Plans

[3] GU,Y., SHI, J.: Introduction to software testing technology (in Chinese) [M], Tsinghua University Press, 2004.

[4] Vector CAST Getting Started [M]. USA, Vector Software Inc., 2006.