Proceedings of the Korean Nuclear Society Autumn Meeting
Taejon, Korea, October 2000

A Study on Design and Testing of Software Module of Safety Software

Sedo Sohn, PoongHyun Seong

Korea Advanced Institute of Science and Technology

Abstract

The design criteria of the software module were based on complexity of the module and the cohesion of the module. The easiness of detection of a fault in the software module can be an additional candidate for the module design criteria. The module test coverage criteria and test case generation is reviewed from the aspects of module testability, easiness of the fault detection. One of the methods is making the numerical results as output in addition to the logical outputs. With modules designed with high testability, the test case generation and test coverage can be made more effective.

1. Introduction

Software life cycle is defined from functional requirements to operation and maintenance phase of the software. In this software life cycle, testing is part of design process. The testing is performed in unit entity, which can be a module and in assembled entity that can be a program with modules combined for a specific functions. Depending on the importance of software, the unit testing phases can be combined into program level testing, or independent personnel not involved in the design of software might perform the testing. Regardless of classification of software, testing is usually considered after the implementations have been finished. But there have been papers recommending to perform design considering easiness of testing, testability, arguing testing can be improved by designing the software with testing considered in advance. Software testing can be classified as white box testing and black box testing. The white box testing is performed at the module level testing. The black box testing is performed at the program level testing which verifies the functional requirements or requirements at the system

level of the software. The white box testing is also called as the program based testing. The white box testing can be classified structure based testing and fault based testing. Test cases of program based testing are generated based on the structure of the program, such as internal data flow or control flow of the software. The structure based testing can be classified into various methods depending on its coverage criteria; define-use test, path testing, branch testing, etc. The fault based testing tries to find errors in the program by making mutation version of the program by inserting the faults at the specified locations. The original and mutated program results are compared each other to evaluate whether the test cases can detect inserted types of faults. The fault based testing can be used to evaluate effectiveness of the other test scheme. The black box testing can be classified as random testing, functional testing, specification based testing, etc. There have been papers comparing effectiveness of each testing scheme over the resources spent to find errors, number of errors found.

The software errors are related to the program complexity, and various software metrics have been proposed for appropriate program complexity. Simple measures are based on measures that can be observed directly form programs, such as number of program lines, number of operators, and number of operands, etc. There are very complicated measures like Halstead's Software Science. Among these software measures, the McCabe cyclomatic complexity metric is used widely. This is based on the assumption that the software complexity is related to the number of control paths generated by the code. From experimental analysis, the programs with cyclomatic complexity greater than 10 is error-prone, and recommended not to exceed that value [1]. Another measure is the Halstead's Software Science which is based on the assumption that a program should be viewed as an expression of language. Halstead's Software Science calculates the measure from the number of distinct operators, number of distinct operands, number of all operators in the code, and the number of all operands in the code [1]. There has been a paper on software complexity analysis with graph theory. The program graph is constructed from the program graph and the graph is analyzed by building adjacency matrix, fundamental circuit matrix that represents the identification of program paths. Program complexity measures are performed from the operation of the matrix operations [2]. One method of fault based testing is the RELAY fault propagation model. The RELAY model of faults and failures is based on incorrect intermediate state from a faulty statement to output. It defines necessary and sufficient conditions for detecting certain classes of faults. REALY models a fault composed of three distinct steps of introduction of an origination of a fault, computational transfer of a fault, propagation of an error to the output. The original state potential failure occurs when the node containing the fault evaluates incorrectly. Even the error state is introduced, certain conditions should be met for an error state to result in failure. For the propagation of an error, both the data flow and control flow is considered. The comprehensive transfer information can provide guide

in testing to investigate whether certain constructs are more prone to coincidental correctness [3].

2. Software Testability

There have been efforts for the testability of the software. Software testability is defined as the probability that a piece of software will fail on its next execution during testing if the software includes a fault [4]. Software testability focuses on the probability that a fault in a program will be revealed by testing. Testability analysis identifies several places in the code where testing is highly unlikely to reveal faults. And further formal analysis and non-random tests are devised to exercise these sections more extensively. If we can change the functional description to include more internal information, we should be able to increase testability upper bound. Software testability can be predicted by sensitivity analysis that seeds the fault and observing the propagation into failures. As design heuristics for testability, J. Voas suggest three methods, specification decomposition to reduce error cancellation, minimize variable reuse, increasing out-parameters. Out-parameters mean specifying special output variables that are specified and implemented specifically and exclusively for testing [4]. For this idea of making software easy to reveal faults, A. Bertolino argues different opinion. For complex software, distributions of faults are not known and faults are due to subtle faults, mismatches between module interfaces, and testability should consider imperfectness of test oracle. We should make program more robust, and increase coverage by improving oracle or observability of internal state, and choose test input distribution to increase error infection probability. Raising testability by raising the failure probability is dangerous, and should not increase failure probability while increasing probability of detecting faults. We need more refined testing tools and need improving the internal error detection capability [5]. Other method of analyzing testability introduces the domain testability with observability and controllability. A program is domain testable if a program is observable and controllable. The observability of the program is defined as the ease of determining if specified inputs affect the outputs, the controllability is defined as the ease of producing a specified output from a specified input. Testing of domain testable specification takes less time than non-domain testable specification. The output of program is function of its input and state, the observability implies that the output value of a program is a function of the input values only. The data characteristics are either explicit test inputs and outputs (specific identifier in program) and implicit test inputs and outputs (component states). Programs can be made more observable by introducing more input based on implicit states, and program controllability can be increased by restricting the range of outputs. To increase the

controllability, the output range should be defined more restrictively [6]. There are efforts measuring the testability early in the design stage based on the specification. The testability of the software can be roughly measured from the software input domain and software output range. If the ratio of the [domain/range] is big, then the testability is low. If the ratio is small, the testability is high [4].

3. Module Testing of the Core Protection Calculator System

The Core Protection Calculator System (CPCS) performs the safety function of generating Departure from Nucleate Boiling Ratio (DNBR) trip and Local Power Density (LPD) trip signals. The software consists of several programs performing its own specific calculations. The UPDATE software is one of the programs and updates the detailed calculation of the DNBR based on the newly read input signals. The UPDATE is composed of several modules that perform specified calculations. Each module is designed with the principles of single entry and single exit, small enough to analyze the branches are exercised by a particular set of input data. The module verification is performed branch coverage testing based on the flow chart. Variable Over-Power Trip (VOPT) module is one of UPDATE modules that generates protection signal when the calculated reactor power is greater than the pre-set value or the increase rate of the reactor power is excessive. The protection signal is generated when the actual power is greater than the calculated trip setpoint. The trip setpoint is a variable, and is calculated to follow the actual power with preset margin within maximum ceiling. But the increase rate of the trip setpoint is limited by pre-specified value. Thus the trip setpoint is changing as the measured actual power is changing, but as the actual power increases at a greater rate than the setpoint can change, the actual power becomes greater than the trip setpoint resulting in trip signal generation. The VOPT is based on auctioneered power from excore detector signals, temperature shadowing corrected excore detector signals, neutron flux power, and thermal power. A variable, FOLLOW, is calculated which follows changes in the auctioneered power within rate limits. FOLLOW cannot be changed from its previous value by more than an amount depending on the data base limits and the computing interval. The VOPT setpoint is computed by adding a fixed power bias, ΔSP_V, to the variable FOLLOW. The VOPT setpoint is limited by a minimum allowable value, SP_{VMIN}, and a maximum allowable value, SP_{VMAX}. The trip is set by adding 40 to the auxiliary trip flag J_{TRP}.

The raw neutron flux power and the temperature corrected neutron flux power are calculated by the following algorithm [7],

$$\Phi_{RAWD} = K_{CAL}(D1 + D2 + D3) \bullet 100.0$$

$$\Phi_{TCORD} = \Phi_{RAWD} \bullet T_F$$

The maximum power is then selected.

$$\Phi_{MAX} = \max(\Phi_{RAWD}, \Phi_{TCORD}, \Phi_{CAL}, B_{\Delta T})$$

Where,

D1,D2,D3 = scaled, excore neutron flux detector values
K_{CAL} = addressable neutron flux power calibration constant
Φ_{CAL} = calibrated neutron flux power(% of rated power)
$B_{\Delta T}$ = calibrated static component of thermal power, % of rated power

Φ_{RAWD} = raw neutron flux power (% rated power)
Φ_{TCORD} = temperature corrected neutron flux power (% rated power)
T_F = coolant temperature shadowing factor
Φ_{MAX} = maximum power (% rated power)

The FOLLOW is calculated by,

$$FUP = \min(\Phi_{MAX}, FOLLOW_P + SUPMAX)$$
$$FDN = FOLLOW_P - SDNMAX$$
$$FOLLOW = \max(FUP, FDN)$$

The VOPT setpoint is then calculated:

$$SP_{VOP1} = \min((FOLLOW + \Delta SP_V), SP_{VMAX})$$
$$SP_{VOPT} = \max(SP_{VOP1}, SP_{VMIN})$$
$$If \Phi_{MAX} \geq SP_{VOPT} \cdot then \cdot J_{TRP} = 40 + J_{TRP}$$

Above, the subscript p denotes value from previous execution

SUPMAX = maximum increase for FOLLOW within an execution of UPDATE(% rated power)

SDNMAX = maximum decrease for FOLLOW within an execution of UPDATE(% rated power)

FOLLOW = rate limited maximum of auctioneered power(% rated power)

FUP, FDN = intermediate variables used to calculated FOLLOW

ΔSP_V =amount in % power, by which VOPT setpoint is above FOLLOW

SP_{VMAX} = maximum allowable value of VOPT setpoint(% rated power)

SP_{VOP1} = intermediate variable used to calculate VOPT setpoint

SP_{VOPT} = VOPT setpoint (% rated power)

J_{TRP} = Auxiliary trip flag

The flow chart for the VOPT module is shown in Figure 1. It shows branches with sequential parts condensed into single description of box. With this flow chart, the test cases can be built to cover all the branches. The Fortran program for the VOPT function except variable definition is shown in Figure 2. The SUPMAX, SDNMAX, and SPVMIN are constants having values of 0.02,

0.5, and 30.0 respectively. All other variables are real valued variable except JTRP which is an integer variable. For this module, 5 test cases are selected to perform the testing which is shown in the Table 1, with additional test case 6. The condition table is drawn to calculate and check proper test sets are generated to cover the entire branch at least once. The expected outputs are shown in the table 2. Even the main output is the auxiliary trip flag, JTRP, there are two more outputs that are used in some other modules.

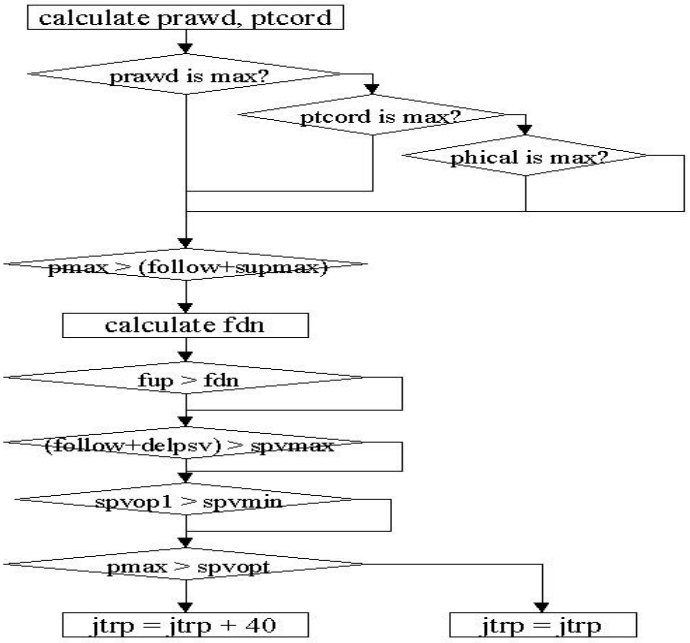

Figure 1 Flow Chart of VOPT

One of the method to increase the testability of software is to make outputs such that the Domain Range Ratio of the module to be close to 1 [4]. By making the numerical output available in addition to the boolean value, the testability can be increased. In case of VOPT function, the output is the JTRP flag that is set as the trip condition occurs. By making outputs FOLLOW and PMAX available in addition to the JTRP, the testability can be increased. With the boolean output JTRP alone, there is chance that faulty error states can not be detected with test cases. But with output FOLLOW and PMAX, the numerical error states can be detected more easily with test cases.

```
 PRAWD        =  KCAL * (D1+D2+D3) * 100.0
 PTCORD       =  PRAWD * TCORF
 PMAX         =  DMAX1 (PRAWD,PTCORD,PHICAL,BDT)
 FUP          =  DMIN1 (PMAX,(FOLLOW+SUPMAX))
 FDN          =  FOLLOW-SDNMAX
 FOLLOW       =  DMAX1 (FUP,FDN)
 SPVOP1       =  DMIN1 ((FOLLOW + DELSPV),SPVMAX)
 SPVOPT       =  DMAX1 (SPVOP1, SPVMIN)
IF (PMAX .GE. SPVOPT) THEN
 JTRP         = 40 + JTRP
 ENDIF
```

Figure 2. VOPT Algorithm Listing

The test was performed with fault seeding by replacing the * operator in line 2 of the program with + operator for test case 6. With this fault seeded, the JTRP are correct as expected, the PMAX value is different. Thus adding additional outputs that can compensate the domain range ratio for the outputs can increase the testability of the program. This results shows similar results as the J. Voas's recommendations to increase the testability by reducing the domain range ratio by inserting additional outputs for testability. The state variable FOLLOW has been also classified as test inputs and outputs, such that the observability of the module is increased resulting in high testability.

Table 1. Test Case for VOPT function

Test case / Variables	1	2	3	4	5	6
PHICAL	95.0	80.0	1001.0	28.0	94.24	95.0
FOLLOW	90.0	90.0	90.0	80.0	10.0	90.0
D1	0.31	0.67	0.30	0.30	0.3543	0.31
D2	0.40	0.12	0.50	0.50	0.3830	0.40
D3	0.30	0.1	0.20	0.20	0.297	0.30
KCAL	1.0	0.01	1.0	0.01	0.999	1.0
TCORF	1.0	1.0	1.1	0.1	0.9957	1.0
BDT	100.0	70.0	1002.0	29.0	93.55	100.0
DELSPV	15.0	15.0	15.0	0.5	15.0	15.0
SPVMAX	110.0	110.0	110.0	30.02	110.0	110.0
JTRP	0	20	0	10	10	0

Table 2. Test Output for VOPT function

Test case / Outut	1	2	3	4	5	6
FOLLOW	90.02	89.5	90.02	79.5	10.02	90.02
PMAX	101.0	80.0	1002.0	29.0	103.33	102.0
JTRP	0	20	40	10	50	0

Some of the modules have many branches. In this case, the generation of the test case is very

time consuming Some of the branches can not be executed by the intermediate calculations not resulting in taking the other branches. In this case, the reason for not taking that branch is investigated and it is justified. The building of the condition table and branch coverage verification takes a lot of time. The verification of the coverage of branches can be automated by checking the machine instruction and its execution branch. The branch coverage criterion is relatively easy criteria to meet, but the define-use coverage and path coverage takes much more efforts. There have been many papers about the appropriate module complexity, such as line number, number of operators, etc. The modularization criteria are cohesion, coupling, complexity, correctness, and correspondence. Cohesion is the glue that holds a module together. The modules can be designed based on functional, sequential, communicational, procedural, temporal, or logical cohesion. The functional cohesion results when every function within the module contributes directly to performing one single function. Where possible, functional, sequential, and communicational strength modules should be given preference over modules with lower levels of cohesion [8]. Coupling is a measure of the strength of interconnection between modules. Other factors are described in the reference 8. The branch instructions can also cause the error cancellation effects, which causes the error being cancelled by the opposite branch giving the same results. Thus for the resource saving and error reduction, the module should be designed to be granular and the complexity of the module to be limited appropriately based on the test coverage criteria to be applied. One candidate for the modularization criteria can be the testability of the module to reduce error hiding at the program and resources for the testing .

4. Conclusion

The module testing with branch coverage criteria is presented with actual modules from safety system software. The fault delectability of the testing can be increased by making the numerical calculation results in addition to the logical outputs. This also makes the testability of the module increased by making the result comparison easy, and reducing the error cancellation effect that can contribute the error hiding probability. The module design criteria was based on complexity and cohesion of the functionality. The module design can be based on the testability of the module. Because the complexity of the module have great influence of the test case selection, the module design considering the testing can reduce the testing efforts.

References

1. R.S. Pressman, Software Engineering, Practitioner's Approach, McGraw Hill, 1992

2. N.F. Schneiderwind, "Application of Program Graphs and Complexity Analysis to Software Development and Testing", IEEE Trans. On Reliability, Vol. R-28, No.3, Aug. 1979

3. D. Richardson et al., "An analysis of Test Data Selection Criteria using the RELAY model of fault detection", IEEE Tran. On S/W Eng., 1993

4. Jeffrey M. Voas, "Software testability: the new verification", IEEE Software, 1995

5. A. Bertolino et al., "On the use of software testability measures for dependability assessment", IEEE Trans. On S/W Eng., 1996

6. R.S. Freedman, "Testability of Software Components", IEEE Tran. On S/W Eng., Vol. 17, No. 6, June 1991

7. KNFC, Functional Design Requirements for Core Protection Calculator System, 1988

8. G. D. Bergland, "A Guided Tour of Program Design Methodologies", IEEE Computer, Oct., 1981