Development of Software Testing Environment for Safety-critical Software Reliability Quantification

Sang Hun LEE[1], Seung Jun LEE[2], Jinkyun PARK[3], Eun-chan LEE[4], Hyun Gook KANG[1]

1. Department of Mechanical, Aerospace and Nuclear Engineering department, Rensselaer Polytechnic Institute (RPI), 110 8th street Troy NY USA, 12180, Republic of Korea (lees35; kangh6@rpi.edu)
2. School of Mechanical, Aerospace and Nuclear Engineering, Ulsan National Institute of Science and Technology (UNIST), 50 UNIST-gil Ulsan Republic of Korea, 44919, Republic of Korea (sjlee420@unist.ac.kr)
3. Integrated Safety Assessment Division, Korea Atomic Energy Research Institute (KAERI), 111 Daedeok-daero 989beon-gil Yuseong-gu, Daejeon Republic of Korea, 34057, Republic of Korea (kshpjk@kaeri.re.kr)
4. Korea Hydro & Nuclear Power Co., Ltd., 1655 Bulguk-ro Gyeongju-si Gyeongsangbuk-do Republic of Korea, 38120, Republic of Korea (eclee@khnp.co.kr)

Abstract: Recently, software has been used within nuclear power plants (NPPs) to digitalize many instrumentation and control (I&C) systems. To guarantee the safety of the NPP, the software reliability of the safety-critical systems must be properly quantified. In this study, the input-profile-based software test method using simulation-based software test-bed is proposed. The simulation-based software test-bed was developed by emulating the microprocessor architecture (CPU register, memory) of a programmable logic controller (PLC) used in NPP safety-critical applications and capturing its behavior at each machine instruction line. The software test cases which represents the possible states of software input and internal variables were developed in consideration of the digital signal processing of the safety-critical PLC as well as the plant thermo-hydraulics data in case of NPP accident. The effectiveness of the proposed safety-critical software test method is demonstrated via a case study for the KNICS RPS BP processor trip logic. Compared with the existing software testing methods, the proposed method can effectively generate the software test sets required for a software exhaustive testing, and reduce the software testing time by avoiding the repeated test for the same software input. Furthermore, the method can be employed to quantify the software reliability of NPP safety-critical I&C applications, and ensure the safe operation of the NPP.
Keyword: Software Reliability, Digital I&C System, Nuclear Power Plant

1 Introduction

With a shift in technology to digital systems due to analog systems approaching obsolescence and to functional advantages of digital systems, existing plants have begun to replace some analog instrumentation and control (I&C) systems, while new plant designs fully incorporate digital systems[1]. Since the dedicated software is used in nuclear power plant (NPP) digital I&C systems, the failure of the safety-critical software failure can induce the common cause failure (CCF) of processor modules in NPP digital I&C system[2][3]. Therefore, the quantification of software reliability plays a very important role in ensuring the safety of the NPP, and the verification of very low software failure probability is crucial regarding the probabilistic risk assessment (PRA) of digitalized NPP.

Previous quantitative software reliability models (QSRMs) include the methods such as software reliability growth model (SRGM), Bayesian belief network (BBN) model, and test-based method[4]. The SRGM is widely used to assess the reliability of software by estimating the increment of reliability as a result of fault removal over time. However, the SRGM is not applicable to safety-critical software because of its high sensitivity in estimating the number of faults to time-to-failure data and due to the lack of sufficient software failure sets in NPP safety-critical applications[5].

The BBN is a method for software reliability assessment that aggregates disparate information on the software, such as software failure data and quality of software lifecycle activities[6][7]. However, the limitation of the BBN model on quantifying software reliability includes developing a credible BBN model that requires identification of a set of complete and independent software attributes, and the qualification of experts to estimate model parameters and qualitative evidence. Due to those limitations, the uncertainty in the software residual faults and failure probability estimates may be very large which makes it difficult to demonstrate the small failure probabilities of safety-critical software[4].

The test-based approach is another method to assess the reliability of NPP safety-critical software. Test-based methods are those employing standard statistical methods to the results of software tests, in the same way as hardware data is analyzed. Some studies relevant to this approach have been conducted in the nuclear field and are mainly divided into two testing methods: (1) black-box testing[8][9][10], and (2) white-box testing[11][12].

The black-box testing method, also known as functional test, considers a software program as a single entity, take random samples from its input space, determine if the outputs are correct, and use the results in standard statistical analyses. However, since the black-box testing is conducted without the knowledge on the program's internal logic or structure, the limitations of black-box testing include the limited coverage and the completeness of the test cases[13].

The white-box testing methods have the advantage in taking into consideration the internal structures (e.g. paths and nodes of software), so that tests can be performed to ensure that certain parts of the software are tested. However, since the white-box testing intends to test all possible paths and nodes of the software, the number of tests that must be carried out for the exhaustive testing is very large[12]; thus, efficient and effective testing framework for the white-box testing of a safety software is required.

Therefore, this study aims at developing simulation-based software test-bed and input-profile-based test cases for effective and exhaustive white-box testing of the NPP safety-critical software. Fig. 1 shows an overview of the proposed software white-box test-based method.

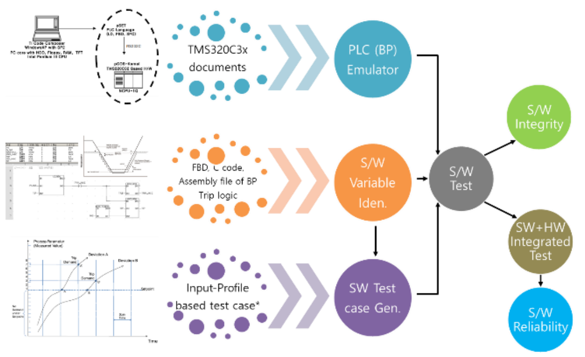


Fig.1 Proposed software input-profile based test method.

Since a programmable logic controller (PLC) widely used in NPP control systems uses a programmable memory to store program instructions and to implement functions such as logic, arithmetic, timing, counting as a binary form[14], the test-bed for software used in reactor protection system (RPS) is developed by emulating its microprocessor architecture of the PLC (e.g. CPU register, memory) and capturing its behavior (e.g. code execution, memory access sequence) at each machine instruction line while conducting its dedicated safety function.

In addition, the software test cases for exhaustive white-box testing are developed based on the software input-profile[11]. The test inputs for the safety-critical application of a nuclear power plant are the inputs which cause the activation of protective action of a NPP (e.g. reactor trip). Since a digital system treats inputs from instrumentation sensors as discrete digital values using an analog-to-digital converter (ADC), the test profile of software input and internal variable can be developed in consideration of those characteristics of NPP digital processing units as well as plant thermo-hydraulics and physics properties during plant transient or design basis accident (DBA).

To demonstrate the effectiveness of the proposed software test method, the test cases for the pressurizer-pressure-low trip logic of RPS bistable processor (BP) of the Korea Nuclear Instrumentation and Control System (KNICS) project[15], are derived and tested on the developed software test environment.

2 Target System

2.1 KNICS IDiPS-RPS BP software

The Integrated Digital Protection System-Reactor Protection System (IDiPS-RPS) is a digitalized reactor protection system developed in the KNICS project for newly constructed NPPs, as well as for upgrading existing analog based RPS[16]. It has the same function as an analog-based RPS to automatically generate a reactor trip signal, and engineered safety features actuation signals whenever process variables reach their corresponding predefined trip set-points.

For an IDiPS-RPS application, the safety-graded PLC platform (POSAFE-Q) was designed[16]. The POSAFE-Q consists of various modules such as a processor module, communication modules, and I/O modules. Especially, the processor module consists of a Texas Instrument C32 DSP CPU and various types of memories (e.g. flash memory, SRAM)[17]. In the memories embedded within the processor module, the IDiPS application software (e.g. BP trip logic, CP voting logic) developed with the POSAFE-Q software engineering tool (pSET) is downloaded. The overview of the RPS software structure is shown in Fig. 2[18].

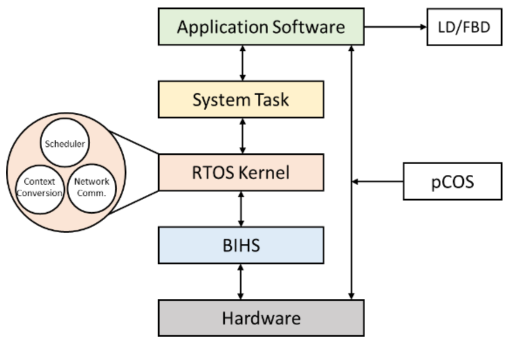


Fig.2 An overview of RPS safety software[18].

As shown in Fig. 2, the application software that will be loaded into a PLC memory is designed based on function block diagram/ladder diagram (FBD/LD) programming[19]. In implementation, the FBD/LD programs and the user-defined function blocks are translated into C programs, and the converted C files are compiled to generate machine instruction code which is loaded to PLC memory area and executed by PLC microprocessors.

2.2 Trip logic of KNICS IDiPS-RPS BP

In KNICS RPS, the BP compares the measured process variables with the predefined trip set-points for determining the reactor trip state. Especially, BP generates a trip signal to the CPs by comparing values of 18 process variables against predefined threshold values[20]. There are four different trip logics built in KNICS BP: (1) fixed set-point trip (10 variables); (2) variable set-point trip (3 variables); (3) manual reset trip (3 variables); and (4) digital trip (2 variables). Table 1 presents the BP trip logics of the prototype KNICS RPS[21] and the description of each trip set-point (TSP) type is as follows:

- Fixed set-point logic: As the process input signal rises or falls through the pre-trip or trip set-point, the BP generates the pre-trip or trip signal, and the trip set-point is decreased by the hysteresis. When BP is un-tripped, it restores the trip set-point.
- Variable set-point logic: BP generates a pre-trip or trip signal when the process input signal reaches the level of the trip or pre-trip set-point, but the set-point value can change depending on the rising or falling of the process input signal and external manual reset, if operator bypass (OB) is permitted.
- Manual reset logic: The BP operation is identical to that of variable set-point logic, but an operator can delay the trip by moving the trip set-point to an upper or lower value by pushing a reset button.
- Digital logic: BP generates a pre-trip or trip signal based on the digital input signal (0 or 1) from other RPS modules such as core protection calculator (CPC).

Table 1 KNICS RPS BP application software modules[21]

Software Modules	Description	OB	TSP type
VA_OVR_PWR_HI Trip (_1_)	Variable Over Power Hi Trip	-	RR, Rising
LOG_PWR_HI Trip (_2_)	Log Reactor Power Hi Trip	Y	Fixed, Rising
LPD_HI Trip (_3_)	Local Power Density Hi Trip	-	Digital
DNBR_LO Trip (_4_)	Departure from Nucleate Boiling Ratio Low Trip	-	Digital
PZR_PR_HI Trip (_5_)	Pressurizer Pressure Hi Trip	-	Fixed, Rising
PZR_PR_LO Trip (_6_)	Pressurizer Pressure Low Trip	Y	MR, Falling
SG1_LVL_LO_RPS Trip (_7_)	SG-1 Low Level Trip	-	Fixed, Falling
SG2_LVL_LO_RPS Trip (_8_)	SG-2 Low Level Trip	-	Fixed, Falling
SG1_LVL_LO_ESF Trip (_9_)	SG-1 Low-Low Level Trip	-	Fixed, Falling
SG2_LVL_LO_ESF Trip (_A_)	SG-2 Low-Low Level Trip	-	Fixed, Falling
SG1_LVL_HI Trip (_B_)	SG-1 Hi Level Trip	-	Fixed, Rising
SG2_LVL_HI Trip (_C_)	SG-2 Hi Level Trip	-	Fixed, Rising
SG1_PR_LO Trip (_D_)	SG-1 Low Pressure Trip	-	MR, Falling
SG2_PR_LO Trip (_E_)	SG-2 Low Pressure Trip	-	MR, Falling
CMT_PR_HI Trip (_F_)	Containment Hi Pressure Trip	-	Fixed, Rising
CMT_PR_HH Trip (_G_)	Containment Hi-Hi Pressure Trip	-	Fixed, Rising
SG1_FLW_LO Trip (_H_)	SG-1 Low Coolant Flow Trip	-	RR, Falling
SG2_FLW_LO Trip (_I_)	SG-2 Low Coolant Flow Trip	-	RR, Falling
CWP Trip (_J_)	CPC-CWP	-	Digital

* Fixed: Fixed Trip Setpoint; MR: Variable Trip Setpoint by Manual Reset; RR: Variable Trip Setpoint by Automatic Rate-Limiting; Digital: On/Off Trip; OB: Operator Bypass;

As previously discussed, the BP trip logics shown in Table 1 are programmed with FBDs. Fig. 3 shows a part of RESET_FALLING logic[21]. The value of output TRIP_LOGIC is generated from the combined execution of several function blocks. The LE_REAL function block in the leftmost position receives the process variable (PV_OUT) and internal variable (TSP) as inputs and computes the output. The result of LE_REAL function block combined with TRIP_LOGIC variable passes and is used in next function blocks (MOVE_BOOL, ADD2_REAL) as inputs. If the result is true, each function block sets TRIP_LOGIC variable as true and increases the value of TSP variable with the value of HYS variable.

The functions of whole BP trip logic are configured by a network of function blocks in the form of a circuit as a function between input variables and output variables. Therefore, the possible states of the input, internal, and output variables can be analyzed based on the software logic and the correlation between each variable.

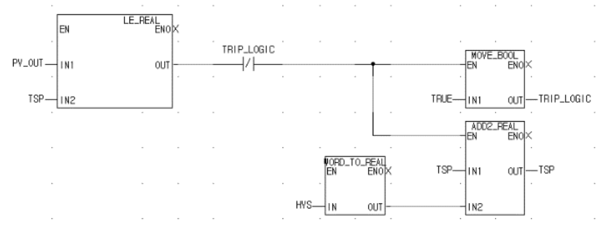


Fig.3 A part of BP RESET_FALLING logic.

3 Methods

3.1 Development of BP software test-bed

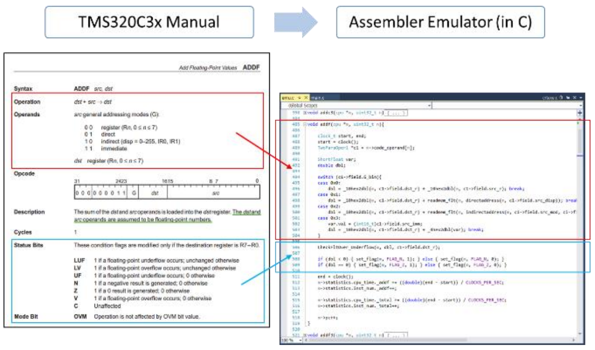
The most fundamental characteristic of PLC operation is their cyclic operation mode[22]. Especially, each iteration of the cyclic operation of PLC, called a scan cycle, consists of several operation stages that are sequentially repeated. After checking its own status, the equipment will copy all the physical input values into its memory, which is called an input scan. Then the output of the software will be updated based on the embedded logics. These operations are repeated at a fixed interval of time, called a scan time.

Therefore, by capturing both the internal (e.g. CPU architecture) and external (e.g. input/output states of program variables) representation of PLC, a PLC microprocessor emulator can be developed which can be used to simulate the behavior of the software for various states of input/internal variable and validate the output for specific software program. In this study, a test-bed for KNICS RPS BP trip logic software was developed in C environment by emulating the microprocessor architecture (e.g. CPU register, memory) of a POSAFE-Q which uses TMS320C32 processor[23] and capture the execution characteristics (e.g. CPU register, memory access sequence) of the BP trip logic at each machine instruction line.

The main part of the test-bed is the CPU of the PLC microprocessor, which contains the CPU internal resources (CPU registers) and the processor's logic, such as arithmetic logic unit (ALU), floating-point/integer multiplier.

The second module of the test-bed is the memory units that is accessible to the CPU, which contains the total memory space of 16M 32-bit words. Within the 16M-word address space, the program, data, and I/O space are contained, allowing the storage of tables, coefficients, program code, or data of the BP trip logic software. In order to simulate reading/writing from/to memory space, several different memory addressing modes (e.g. register, direct, indirect, immediate) were implemented in the test-bed.

After implementing the CPU architectures into the test-bed, a total of 113 TMS320C3x instruction sets were implemented in the test-bed to emulate the instruction execution at each PLC scan cycle. All instructions are defined as a single machine word long (32-bit), and most instructions require one cycle to be executed. The instruction sets contain the instructions for load and store, 2-/3-operand arithmetic, program control, interlocked operations, and parallel operations. The syntax of instructions contains their specific 9-bit opcodes, and the addressing mode and operands are defined for each instruction. Based on the instruction execution, the contents in the CPU registers, memory, and system stack are changed, and the conditional flags stored in the CPU status register are updated by the result of each instruction. In consideration of the characteristics of each instruction set, the function of each instruction set was written in C code and integrated within the test-bed, as shown in Fig. 4.



Add Floating-Point Values **ADDF**

Syntax	**ADDF** *src, dst*
Operation	*dst* + *src* → *dst*
Operands	*src* general addressing modes (G): 0 0 register (R*n*, 0 ≤ *n* ≤ 7) 0 1 direct 1 0 indirect (disp = 0–255, IR0, IR1) 1 1 immediate *dst* register (R*n*, 0 ≤ *n* ≤ 7)
Opcode	31–24: 0 0 0 0 0 0 0 1 1 (23) | G | *dst* (16–15…8) | *src* (7–0)
Description	The sum of the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.
Cycles	1
Status Bits	These condition flags are modified only if the destination register is R7–R0. **LUF** 1 if a floating-point underflow occurs; unchanged otherwise **LV** 1 if a floating-point overflow occurs; unchanged otherwise **UF** 1 if a floating-point underflow occurs; 0 otherwise **N** 1 if a negative result is generated; 0 otherwise **Z** 1 if a 0 result is generated; 0 otherwise **V** 1 if a floating-point overflow occurs; 0 otherwise **C** Unaffected
Mode Bit	**OVM** Operation is not affected by OVM bit value.

```
void addi3(cpu *n, uint32_t n){ ... }

void addf(cpu *n, uint32_t n){

    clock_t start, end;
    start = clock();
    TwoParaOper1 *c1 = n->code_operand[n];

    ShortFloat var;
    double dbl;

    switch (c1->field.G_bin){
    case 0x0:
        dbl = _10hex2dbl(n, c1->field.dst_r) + _10hex2dbl(n, c1->field.src_r); break;
    case 0x1:
        dbl = _10hex2dbl(n, c1->field.dst_r) + readmem_flt(n, directaddress(n, c1->field.src_disp)); brea
    case 0x2:
        dbl = _10hex2dbl(n, c1->field.dst_r) + readmem_flt(n, indirectaddress(n, c1->field.src_mod, c1->f
    case 0x3:
        var.val = (int16_t)c1->field.src_imm;
        dbl = _10hex2dbl(n, c1->field.dst_r) + _4hex2dbl(var); break;
    }

    checkfltover_underflow(n, dbl, c1->field.dst_r);

    if (dbl < 0) { set_flag(n, FLAG_N, 1); } else { set_flag(n, FLAG_N, 0); }
    if (dbl == 0) { set_flag(n, FLAG_Z, 1); } else { set_flag(n, FLAG_Z, 0); }

    end = clock();
    n->statistics.cpu_time._addf += ((double)(end - start)) / CLOCKS_PER_SEC;
    n->statistics.inst_num._addf++;

    n->statistics.cpu_time._total += ((double)(end - start)) / CLOCKS_PER_SEC;
    n->statistics.inst_num._total++;

    n->pc++;
}

void addf3(cpu *n, uint32_t n){ ... }
```

Fig.4 Implementation of TMS320C3x instruction sets within software test-bed.

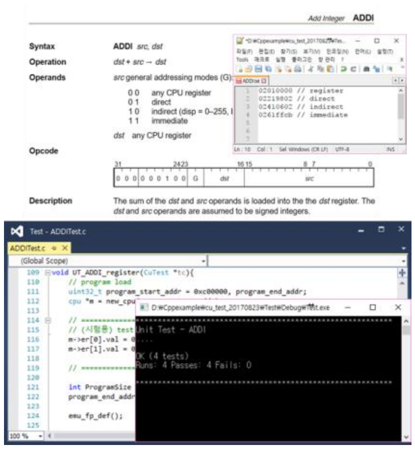
In order to conduct a finite exhaustive test of KNICS BP trip logic using the developed test-bed, the program file of the BP trip logic and the constant file which contains the memory map of the input (e.g. pressure, water level) and internal variable (e.g. counter, test parameter) used in the BP trip logic were loaded to the test-bed, and the output file after the program execution (i.e. after one scan time of PLC) given a specific software input file was automatically created and the specific memory area related to safety signal output (e.g. trip/pre-trip signal) were checked to verify the output of the test-bed.

3.2 Verification of the BP software test-bed

In order to validate the correctness of the developed simulation-based software test-bed, both the unit testing and functional testing for the software test-bed were conducted.

3.2.1. Unit testing of BP software test-bed

The unit testing is a software testing method by which individual units of the source code, such as associated control data, usage procedures, and operating procedures, are tested to determine whether each unit of the code generates correct output[24]. In this study, the Cutest framework[25], a C unit testing framework, was used to write and run the unit tests for each instruction sets implemented in the developed test-bed. The unit test cases for each instruction set were developed in consideration of all possible addressing modes and operands, and validated whether all code areas were correctly executed by verifying the result of the generated test cases on the developed test-bed by comparing with its expected results as shown in Fig. 5.



Add Integer **ADDI**

Syntax	**ADDI** *src, dst*
Operation	*dst + src → dst*
Operands	*src* general addressing modes (G):
	0 0 any CPU register
	0 1 direct
	1 0 indirect (disp = 0–255, I
	1 1 immediate
	dst any CPU register
Opcode	31–24: 0 0 0 0 0 0 1 0 0; 23: G; 16–15: *dst*; 8–0: *src*
Description	The sum of the *dst* and *src* operands is loaded into the the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

```
02010000 // register
02219802 // direct
02410602 // indirect
0261ffcb // immediate
```

Test - ADDITest.c

```
void UT_ADDI_register(CuTest *tc){
    // program load
    uint32_t program_start_addr = 0xc00000, program_end_addr;
    cpu *m = new_cpu

    // ============
    // (시험용) test
    m->er[0].val = 0
    m->er[1].val = 0

    // ============

    int ProgramSize
    program_end_addr

    emu_fp_def();
```

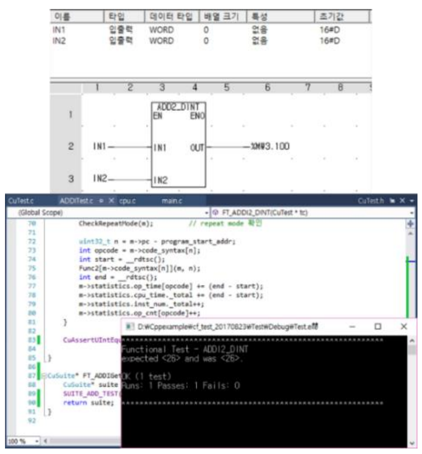
D:₩Cppexample₩cu_test_20170823₩Test₩Debug₩Test.exe

```
*****************************************************************
Unit Test - ADDI
....

OK (4 tests)
Runs: 4 Passes: 4 Fails: 0

*****************************************************************
```

Fig.5 An example of instruction set testing of test-bed.

3.2.2. Functional testing of BP software test-bed

Functional testing is a type of black-box testing for the source code that are tested by checking the correctness of the program by comparing the results for a given specific input. In this study, the generated machine languages of the function block diagrams defined in IEC61131-3[26] (e.g. ADD, AND, EQ) using TMS320C3x compiler are used to test the functionality and correctness of the output of the developed test-bed, as shown in Fig. 6.



이름	타입	데이터 타입	배열 크기	특성	초기값
IN1	입출력	WORD	0	없음	16#D
IN2	입출력	WORD	0	없음	16#D

FT_ADDI2_DINT(CuTest * tc)

```
        CheckRepeatMode(m);          // repeat mode 확인

        uint32_t n = m->pc - program_start_addr;
        int opcode = m->code_syntax[n];
        int start = __rdtsc();
        Func2[m->code_syntax[n]](m, n);
        int end = __rdtsc();
        m->statistics.op_time[opcode] += (end - start);
        m->statistics.cpu_time._total += (end - start);
        m->statistics.inst_num._total++;
        m->statistics.op_cnt[opcode]++;
    }

    CuAssertUIntEqu

}

CuSuite* FT_ADDIGet
    CuSuite* suite
    SUITE_ADD_TEST(
    return suite;
}
```

D:₩Cppexample₩cf_test_20170823₩Test₩Debug₩Test.exe

```
*****************************************************************
Functional Test - ADDI2_DINT
expected <26> and was <26>.

OK (1 test)
Runs: 1 Passes: 1 Fails: 0

*****************************************************************
```

Fig.6 An example of function block testing of test-bed.

4 Case Study

4.1 Test case generation of BP trip logic

The generation of software test cases by mapping the software for all its possible transition states is one of the key steps in software test-based method. Previous test-based approaches[8][9] conducted in the nuclear field involves the development of an input set for a software test as a trajectory form (a series of successive values for the input variables of a program that occur during the operation of the software over time) by random sampling the test sets from the input profile. However, the limitation of those approaches involves the uncertainty caused by random sampling, the ambiguity on the necessary length of a trajectory, and a long execution time per test case.

Since the software failure is basically a deterministic process, i.e. the software will follow the same execution path and generate the same output for the same input, it is possible to test software using input set composed of a combination of single values of each software input and internal variable. When the finite domains for each software input and internal sets are identified, the output of the software can be captured for each input/internal sets, thus there is no need to form a trajectory form of input. This allows the test execution time to be drastically reduced, and the total number of tests which covers all possible software states during its operation can be expressed mathematically.

In this study, the software test cases for BP trip logic was developed by deriving the possible input/internal domain of the software. The test inputs for the NPP safety-critical applications (e.g. RPS) are the inputs which cause the activation of protective action such as a reactor trip signal generation. Since a digital I&C system in NPP treats inputs from instrumentation sensors as discrete digital values using an ADC, software input profile can be developed in consideration of the characteristics of digital components as well as the plant physical and thermo-hydraulic properties. In case of software internal variable, its profile can be developed based on the possible range of each internal variable and the software internal logic.

Table 2 Summarized variables for PZR_PR_LO (_6_) trip logic test case generation

Variable	Description	Format	Type*
T_SCAN_FLAG	Flag for PLC scan operation (operation/test)	BOOL	SV
BP_INTEST	BP test status	BOOL	SV
_6_PTSP_R	PZR_PR_LO pre-trip set-point	WORD	SV
_6_TSP_R	PZR_PR_LO pre-trip set-point	WORD	SV
_6_RST_DELAY_CNT_R	PZR_PR_LO reset delay count	WORD	SV
_6_OB_PERM	Operator trip bypass permission	BOOL	IV
_6_OB_REQ_MCR	Operator trip bypass request (from MCR)	BOOL	IV
_6_OB_REQ_RSR	Operator trip bypass request (from RSR)	BOOL	IV
_6_RST_REQ_MCR_DI	Trip set-point reset signal (from MCR)	BOOL	IV
_6_RST_REQ_RSR_DI	Trip set-point reset signal (from RSR)	BOOL	IV
PAT_START	Periodic automatic test start signal	WORD	IV
_6_PV_OUT_AI	PZR_PR_LO process parameter (PZR pressure)	WORD	IV

* SV: State (or internal) variable; IV: Input Variable;

Among various trip logics, the pressurizer-pressure-low (PZR_PR_LO) trip logic was chosen as a case study for developing the test cases since it has more complicated logic (e.g. operator bypass function, reset delay timer, set-point reset by operator), thus has more cases to be tested compared to other fixed- and variable-type trip logics. First, the variables regarding the PZR_PR_LO trip logic were investigated. There are a total of 143 variables in this logic. When variables for constant and temporary variables that are automatically calculated based on input values between scan intervals are excluded, the remaining important variables for reactor trip signal generation can be summarized as shown in Table 2.

Considering the resolution of the ADC used in RPS, the possible states of the PZR_PR_LO trip logic variables can be expressed by combining the possible states of the trip set-point and the reset delay time, and so on. The possible input sets can be expressed by combining the possible states of the input variables (current pressure, bypass from MCR or RSR, reset from MCR or RSR) which can be derived based on the plant thermo-hydraulic analysis and possible deviation of each variable in PLC scan interval (e.g. 50 ms). In this study, a large loss of coolant accident (LOCA) which is one of the fastest transients among the possible deviations regarding NPP reactor coolant system (RCS) pressure drop cases was considered as a target plant DBA for reactor trip signal generation, and the possible states of software input set (process variable) were derived based on the maximum pressure deviation before reactor trip signal obtained using the Multi-dimensional Analysis of Reactor Safety (MARS) code[27], developed in KAERI.

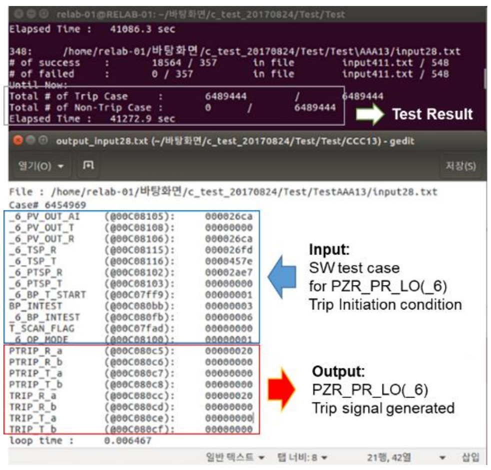
In result, a total of 116,666,784 test cases were derived by combining the possible combinations of both input and internal variables of the BP trip logic shown in Table 2, and the test cases were used as an input to the developed software test-bed to analyze whether the output variable of the BP trip logic software in the memory area was updated correctly (e.g. trip signal generated in trip initiation condition).

4.2 Test results of BP PZR_PR_LO trip logic

Based on the derived test cases for PZR_PR_LO trip logic, the test cases which represent the reactor trip initiation condition were tested using the developed test-bed. The test was conducted in 12.57 hours using sixteen 3.60 GHz logical processors (6.205 ms per test case). From the analysis, the followings were observed:

- The BP trip logic software consists of 32,566 lines of assembler lines and 98,755 lines were executed in average for a single test case. Among the executed instruction sets, LDIU (load integer unconditionally) and LDI (load integer) instructions were executed most frequently (44731 and 14666 times, respectively).

- It was observed that 50.32% and 8.9% of the execution time were spent, and the internal CPU clock used per instruction were 303 and 163 for the LDIU and LDI instructions, respectively in the developed software test-bed. The longest internal CPU clocks used per instruction included instructions related to floating-point operations (e.g. CMPF (2406 clocks), LDFU (1104 clocks)).
- The test results showed that all test cases generated the pre-trip/trip signals for PZR_PR_LO (i.e. _6_TRIP_R = 0x1, _6_PTRIP_R = 0x1) as well as the final pre-trip/trip signals of BP (i.e. PTRIP_R_a = 0x1, TRIP_R_a = 0x1) which will be sent to CP for the trip signal voting logic. Fig. 7 shows a part of test results conducted for trip initiation condition of PZR_PR_LO trip logic.


```
relab-01@RELAB-01: ~/바탕화면/c_test_20170824/Test/Test
Elapsed Time :   41086.3 sec

348:     /home/relab-01/바탕화면/c_test_20170824/Test/Test\AAA13/input28.txt
# of success    :       18564 / 357       in file       input411.txt / 548
# of failed     :       0 / 357           in file       input411.txt / 548
Until Now:
Total # of Trip Case     :       6489444         /       6489444
Total # of Non-Trip Case :       0       /       6489444
Elapsed Time :   41272.9 sec
```

```
output_input28.txt (~/바탕화면/c_test_20170824/Test/Test/CCC13) - gedit
File : /home/relab-01/바탕화면/c_test_20170824/Test/TestAAA13/input28.txt
Case# 6454969
_6_PV_OUT_AI     (@00C08105):     000026ca
_6_PV_OUT_T      (@00C08108):     00000000
_6_PV_OUT_R      (@00C08106):     000026ca
_6_TSP_R         (@00C08115):     000026fd
_6_TSP_T         (@00C08116):     0000457e
_6_PTSP_R        (@00C08102):     00002ae7
_6_PTSP_T        (@00C08103):     00000000
_6_BP_T_START    (@00C07ff9):     00000001
BP_INTEST        (@00C080bb):     00000003
_6_BP_INTEST     (@00C080fb):     00000006
T_SCAN_FLAG      (@00C07fad):     00000000
_6_OP_MODE       (@00C08100):     00000001
PTRIP_R_a        (@00C080c5):     00000020
PTRIP_R_b        (@00C080c6):     00000000
PTRIP_T_a        (@00C080c7):     00000000
PTRIP_T_b        (@00C080c8):     00000000
TRIP_R_a         (@00C080cc):     00000020
TRIP_R_b         (@00C080cd):     00000000
TRIP_T_a         (@00C080ce):     00000000
TRIP_T_b         (@00C080cf):     00000000
loop time :      0.006467
```

Fig.7 A part of test results of BP PZR_PR_LO trip logic.

5 Conclusion

In this study, the input-profile-based software test method utilizing simulation-based software test-bed was proposed. The test-bed for software white-box testing was developed considering the characteristics of machine language used by the safety-critical PLC and the CPU architecture of the PLC microprocessor.

Since the software test inputs for the safety-critical application such as RPS of a NPP are the inputs which cause the activation of protective action, such as a reactor trip, the software input profile was developed in consideration of the digital signal processing features of the PLC as well as the plant thermo-hydraulics and physics data in case of plant transients or DBAs. As an application of the proposed software test method, a KNICS RPS BP software logic used in the trip signal generation of was tested based on the machine code of the BP trip logic and the software test cases developed for PZR_LO_PR trip.

An important characteristic of the proposed software test approach is that the test sets can be quantitatively derived to achieve exhaustive testing of the safety-critical software. In addition, it can effectively reduce the software testing time per test case by emulating the software behavior given the software input/internal states in machine language level, compared to the black-box testing which uses trajectory inputs for software testing.

The proposed input-profile based software test method is expected to support the software reliability quantification in NPP safety-critical I&C applications and further ensure the safety of the software.

Nomenclature

ADC	Analog-To-Digital Converter
ALU	Arithmetic Logic Unit
BBN	Bayesian Belief Network
BP	Bistable Processor
CCF	Common Cause Failure
CPC	Core Protection Calculator
CPU	Central Processing Unit
DBA	Design Basis Accident
DSP	Digital Signal Processing
FBD	Function Block Diagram
IDIPS-RPS	Integrated Digital Protection System Reactor Protection System
I&C	Instrumentation and Control
KNICS	Korea Nuclear Instrumentation and Control System
LD	Ladder Diagram
LOCA	Loss of Coolant Accident
MARS	Multi-Dimensional Analysis of

	Reactor Safety
MCR	Main Control Room
NPP	Nuclear Power Plant
PLC	Programmable Logic Controller
PRA	Probabilistic Risk Assessment
PZR	Pressurizer
pSET	POSAFE-Q Software Engineering Tool
QSRM	Quantitative Software Reliability Model
RCS	Reactor Coolant System
RPS	Reactor Protection System
RSR	Remote Shutdown Room
SRGM	Software Reliability Growth Model
TSP	Trip Set-Point

Acknowledgement

This work was supported by the project of 'Evaluation of human error probabilities and safety software reliabilities in digital environment (L16S092000),' which was funded by the Central Research Institute (CRI) of the Korea Hydro and Nuclear Power (KHNP) company.

References

[1] J.-G. Choi and D.-Y. Lee, "Development of RPS Trip Logic Based on PLD Technology," Nuclear Engineering and Technology, vol. 44, no. 6, pp. 697–708, 2012.

[2] H. G. Kang and T. Sung, "An Analysis of Safety-Critical Digital Systems for Risk-Informed Design," Reliability Engineering & System Safety, vol. 78, no. 3, pp. 307–314, 2002.

[3] H.-G. Kang, M.-C. Kim, S.-J. Lee, H.-J. Lee, H.-S. Eom, J.-G. Choi, and S.-C. Jang, "An Overview of Risk Quantification issues for Digitalized Nuclear Power Plant using a Static Fault Tree," Nuclear Engineering and Technology, vol. 41, no. 6, pp. 849–858, 2009.

[4] T. L. Chu, M. Yue, M. Martinez-Guridi, and J. Lehner, Review of Quantitative Software Reliability Methods, Brookhaven National Laboratory, 2010.

[5] M.-C. Kim, S.-C. Jang, and J.-J. Ha, "Possibilities and Limitations of Applying Software Reliability Growth Models to Safety-Critical Software," Nuclear Engineering and Technology, vol. 39, no. 2, pp. 129–132, 2007.

[6] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using Bayesian nets," Information and Software Technology, vol. 49, no. 1, pp. 32–43, 2007.

[7] H. Eom, G. Park, S. Jang, H. S. Son, and H. G. Kang, "V&V-based remaining fault estimation model for safety–critical software of a nuclear power plant," Annals of Nuclear Energy, vol. 51, pp. 38–49, 2013.

[8] J. May, G. Hughes, and A. D. Lunn, "Reliability estimation from appropriate testing of plant protection software," Software Engineering Journal, vol. 10, no. 6, p. 206, 1995.

[9] T.-L. Chu, Development of Quantitative Software Reliability Models for Digital Protection Systems of Nuclear Power Plants, Nuclear Regulatory Commission, 2013.

[10] S. Kuball and J. H. R. May, "A discussion of statistical testing on a safety-related application," Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability, vol. 221, no. 2, pp. 121–132, 2007.

[11] H. G. Kang, H. G. Lim, H. J. Lee, M. C. Kim, and S. C. Jang, "Input-profile-based software failure probability quantification for safety signal generation systems," Reliability Engineering & System Safety, vol. 94, no. 10, pp. 1542–1546, 2009.

[12] S. M. Shin, S. H. Lee, H. G. Kang, H. S. Son, S. J. Lee, "Test Based Reliability Quantification Method for a Safety Critical Software using Finite Test Sets", Proceedings of the 9th International Topical Meeting on Nuclear Plant Instrumentation, Control & Human–Machine Interface Technologies (NPIC & HMIT 2015), Charlotte, NC, February 22~26, 2015.

[13] C. V. Ramamoorthy and W.-T. Tsai, "Advances in software engineering," Computer, vol. 29, no. 10, pp. 47–58, 1996.

[14] K. Yau, "PLC Forensics Based on Control Program Logic Change Detection," Journal of Digital Forensics, Security and Law, 2015.

[15] J. H. Park, D. Y. Lee, C. H. Kim, "Development of KNICS RPS Prototype", Proceedings of ISOFIC 2005, Session 6, pp.160-161, Tongyeong, Korea, Nov. 1~4, 2005.

[16] K.-C. Kwon and M.-S. Lee, "Technical Review on the Localized Digital Instrumentation and Control Systems," Nuclear Engineering and Technology, vol. 41, no. 4, pp. 447–454, May 2009.

[17] M.-K. Lee, S.-W. Song, and D.-H. Yun. Development and Application of POSAFE-Q PLC Platform. IAEA-CN-194. 2012.

[18] H. S. Sohn, et al., High-Reliable PLC RTOS Development and RPS Structure Analysis, Korea Atomic Energy Research Institute, 2008.

[19] K. Koo, et al., "Development of Application Programming Tool for Safety Grade PLC (POSAFE-Q)," Transactions of the Korean Nuclear Society Spring Meeting, Chuncheon, Korea, May 25~26, 2006.

[20] J. Yoo, J.-H. Lee, and J.-S. Lee, "A Research on Seamless Platform Change of Reactor Protection System from PLC to FPGA," Nuclear Engineering and Technology, vol. 45, no. 4, pp. 477–488, 2013.

[21] BP SDS for Reactor Protection System, KNICS-

RPS-SDS231 Rev.03, Doosan Heavy Industries and Construction Co., Ltd, 2008.

[22] J. Palomar and R. Wyman, The Programmable Logic Controller and its Application in Nuclear Reactor Systems, Nuclear Regulatory Commission, 1993.

[23] TMS320C3x User's Guide, Texas Instrument, 1997.

[24] D. Huizinga and A. Kolawa. Automated defect prevention: best practices in software management. John Wiley & Sons, 2007.

[25] A. Jalis, CuTest: C Unit Testing Framework, Ver. 2.1.2, April, 2013; http://cutest.sourceforge.net/.

[26] Programmable controllers - Part 3: Programming Languages (IEC 61131-3), International Electrotechnical Commission, 1993.

[27] J.-J. Jeong, K. S. Ha, B. D. Chung, and W. J. Lee, "Development of a multi-dimensional thermal-hydraulic system code, MARS 1.3.1," Annals of Nuclear Energy, vol. 26, no. 18, pp. 1611–1642, Dec. 1999.