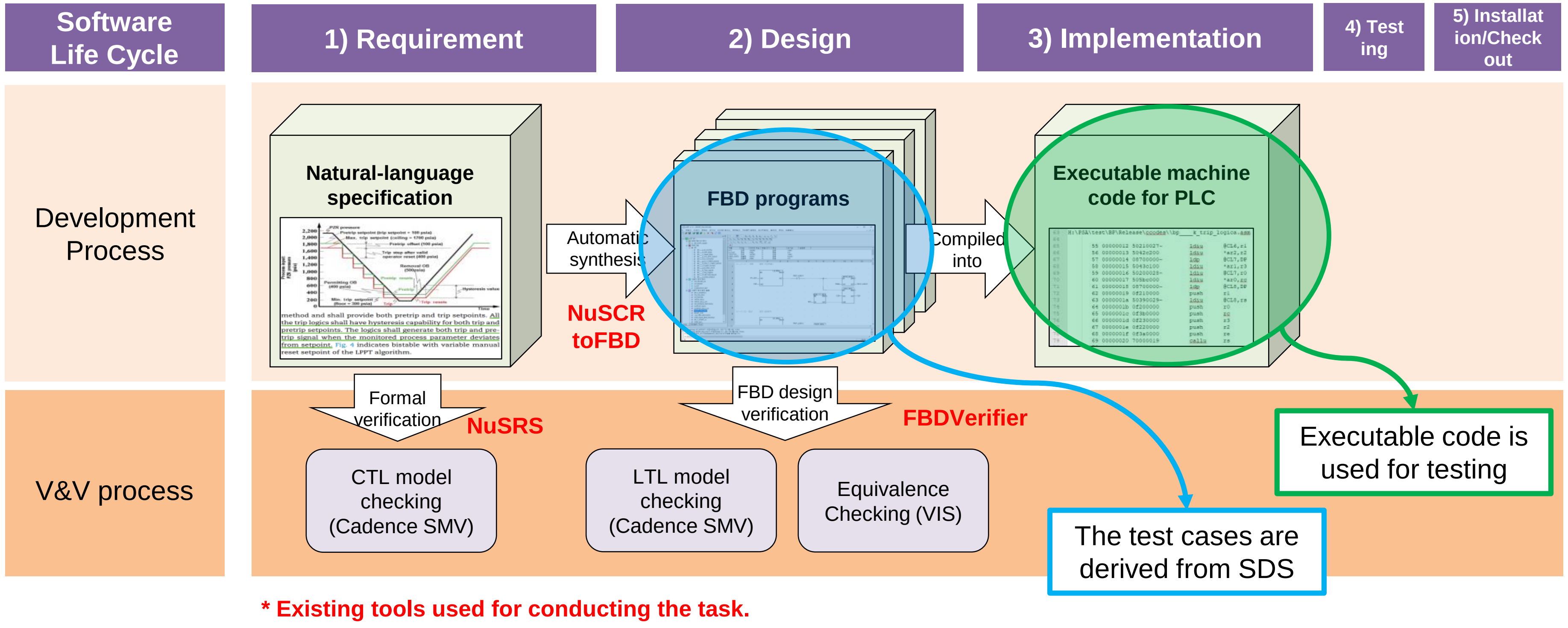
APPENDIX

Appendix I.

- The scope of the method development in this research is limited to modeling NPP software failures for 'failure on-demand' scenario (focus of NPP PRA model).
- The method assumes FBD programs exactly represent the initial software requirements; thus, test cases generated from FBD program reflect all the on-demand situations of that NPP safety software.

Software life cycle and related development/verification processes of a typical NPP safety software

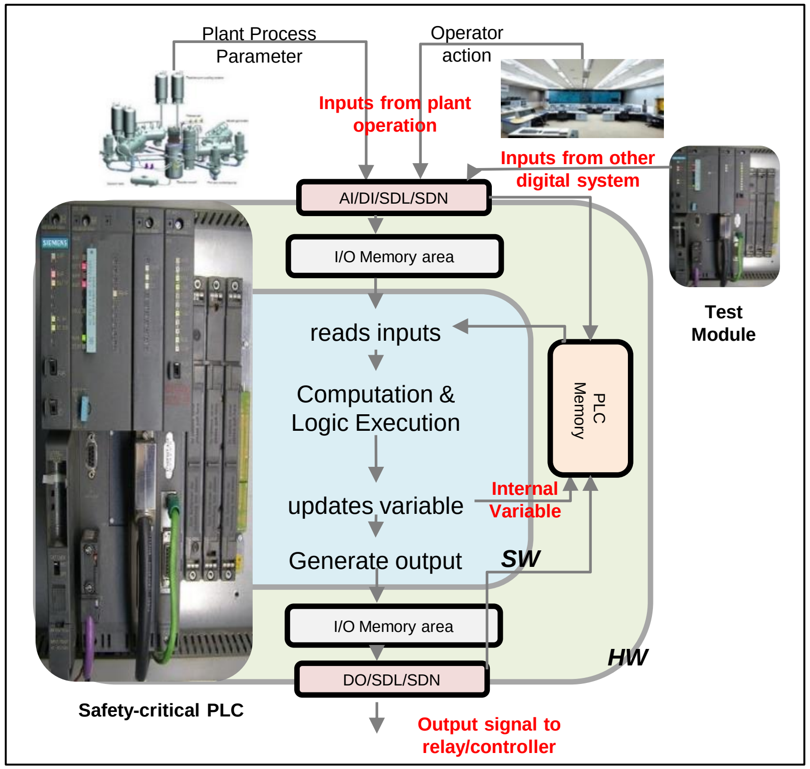

Software Life Cycle
1) Requirement
2) Design
3) Implementation
4) Testing
5) Installation/Check out
Development Process
Natural-language specification
Automatic synthesis
NuSCR toFBD
FBD programs
Compiled into
Executable machine code for PLC
V&V process
Formal verification
NuSRS
CTL model checking (Cadence SMV)
FBD design verification
FBDVerifier
LTL model checking (Cadence SMV)
Equivalence Checking (VIS)
The test cases are derived from SDS
Executable code is used for testing

*** Existing tools used for conducting the task.**

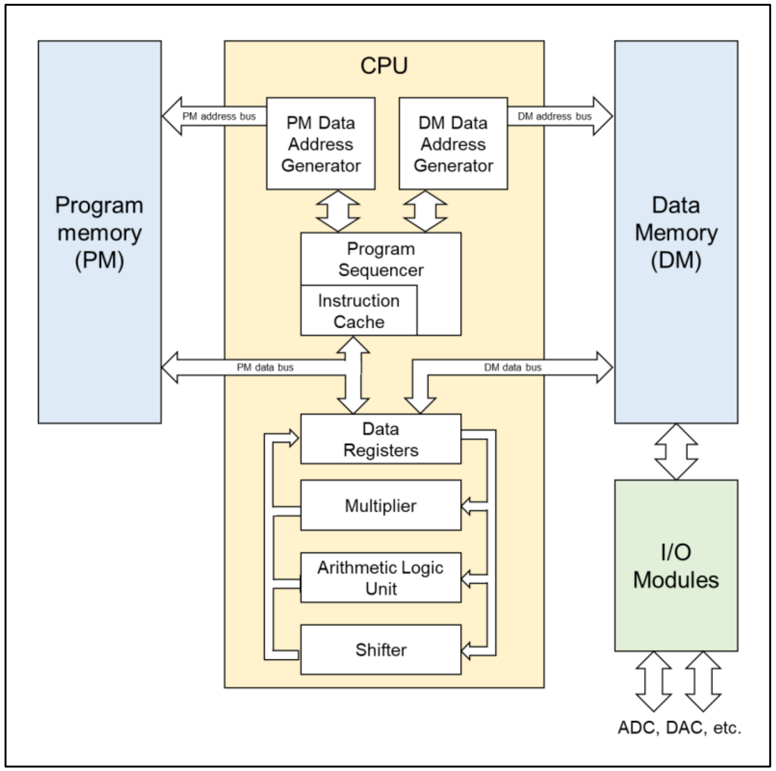
* FBD : function block diagram / PRA : probabilistic risk assessment / CTL : Computation tree logic / LTL : Linear temporal logic / VIS : Verification Interacting with Synthesis / SMV : Symbolic model verifier / SDS : software design specifications (e.g. design documents, FBD files)

Appendix II.

- Basic operation of NPP Safety-graded PLC
  - 1) **Fetch**: the executable code in the memory map is fetched.
  - 2) **Decode**: the fetched code is decoded into a specific instruction set.
  - 3) **Read**: the address is generated and the operands are read from the registers.
  - 4) **Execute**: the operation of the decoded instruction set is performed and the operation results are stored in the CPU register or the memory.



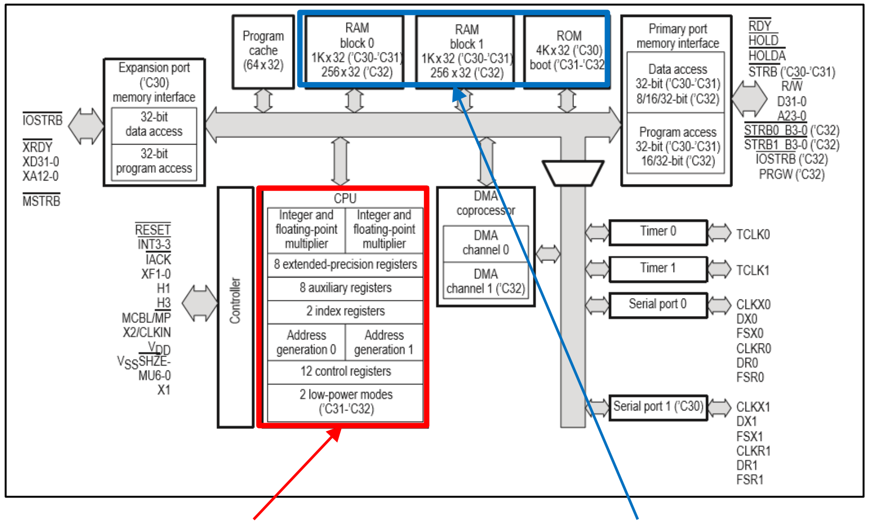
Operating mechanism of a Typical NPP safety PLC



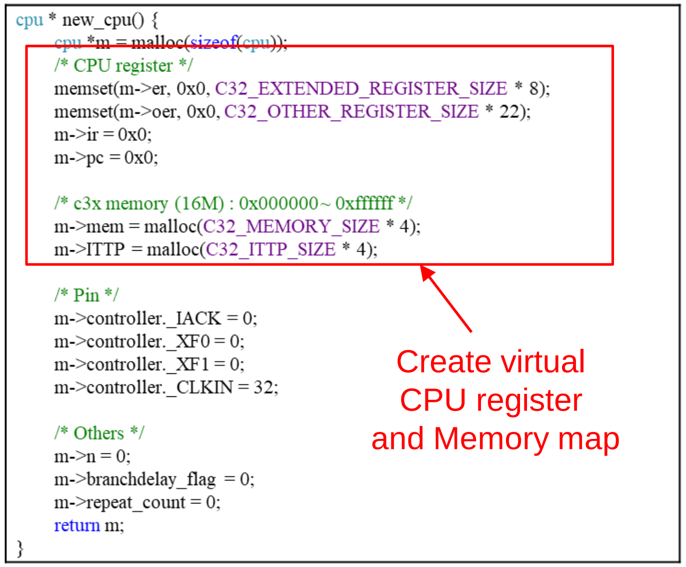
Block diagram of a typical PLC CPU architecture

Appendix III.

- **Development of NPP safety software test-bed (*NSTM*)**
  - 1) Emulation of microprocessor architecture of POSAFE-Q PLC
    - CPU registers (30 registers)
      - Extended-precision registers – extended-precision floating-point expression
      - Auxiliary registers – indirect addressing, 32-bit integer/logical expression
      - Other registers : system functions (e.g., stack, condition, block repeat)
    - Memory map (16 Mbyte; 0x000000 ~ 0xFFFFFF)
      - 16Mbyte 32-bit words of program, data, I/O space
      - Accessible by various addressing modes (e.g. direct, indirect, immediate)



TMS320C3x CPU registers and Memory map

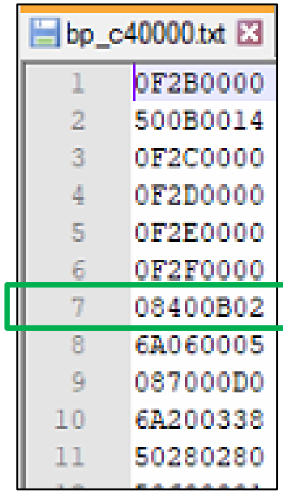


```
cpu * new_cpu() {
        cpu *m = malloc(sizeof(cpu));
        /* CPU register */
        memset(m->er, 0x0, C32_EXTENDED_REGISTER_SIZE * 8);
        memset(m->oer, 0x0, C32_OTHER_REGISTER_SIZE * 22);
        m->ir = 0x0;
        m->pc = 0x0;

        /* c3x memory (16M) : 0x000000~ 0xffffff */
        m->mem = malloc(C32_MEMORY_SIZE * 4);
        m->ITTP = malloc(C32_ITTP_SIZE * 4);

        /* Pin */
        m->controller._IACK = 0;
        m->controller._XF0 = 0;
        m->controller._XF1 = 0;
        m->controller._CLKIN = 32;

        /* Others */
        m->n = 0;
        m->branchdelay_flag = 0;
        m->repeat_count = 0;
        return m;
}
```

CPU and memory emulation in test-bed (C code)

Appendix III.

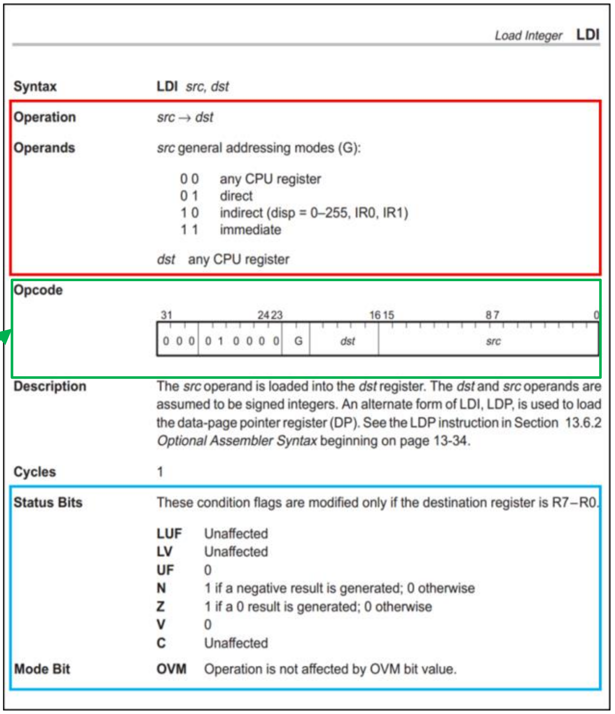
- **Development of NPP safety software test-bed (*NSTM*)**
  - 2) Implementation of microprocessor (TMS320c3x) instruction sets
    - Total of 118 assembly language instructions
    - Types of instruction set:
      - 1) Load and store
      - 2) 2-operand arithmetic/logical
      - 3) 3-operand arithmetic/logical
      - 4) Program control
      - 5) Interlocked operations
      - 6) Parallel operations



bp_c40000.txt

```
0F2B0000
500B0014
0F2C0000
0F2D0000
0F2E0000
0F2F0000
08400B02
6A060005
087000D0
6A200338
50280280
```

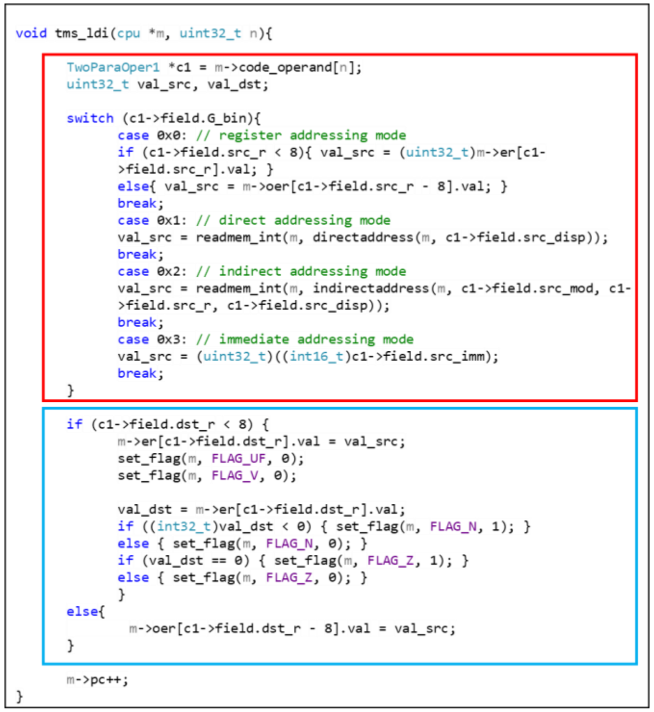
Decode machine code



Load Integer **LDI**

Syntax	**LDI** *src, dst*
Operation	*src* → *dst*
Operands	*src* general addressing modes (G): 0 0 any CPU register 0 1 direct 1 0 indirect (disp = 0–255, IR0, IR1) 1 1 immediate *dst* any CPU register
Opcode	31 … 24 23 … 16 15 … 8 7 … 0 0 0 0 | 0 1 0 0 0 0 | G | *dst* | *src*
Description	The *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers. An alternate form of LDI, LDP, is used to load the data-page pointer register (DP). See the LDP instruction in Section 13.6.2 *Optional Assembler Syntax* beginning on page 13-34.
Cycles	1
Status Bits	These condition flags are modified only if the destination register is R7–R0. **LUF** Unaffected **LV** Unaffected **UF** 0 **N** 1 if a negative result is generated; 0 otherwise **Z** 1 if a 0 result is generated; 0 otherwise **V** 0 **C** Unaffected
Mode Bit	**OVM** Operation is not affected by OVM bit value.

Description of **LDI (load integer)** instruction set operation

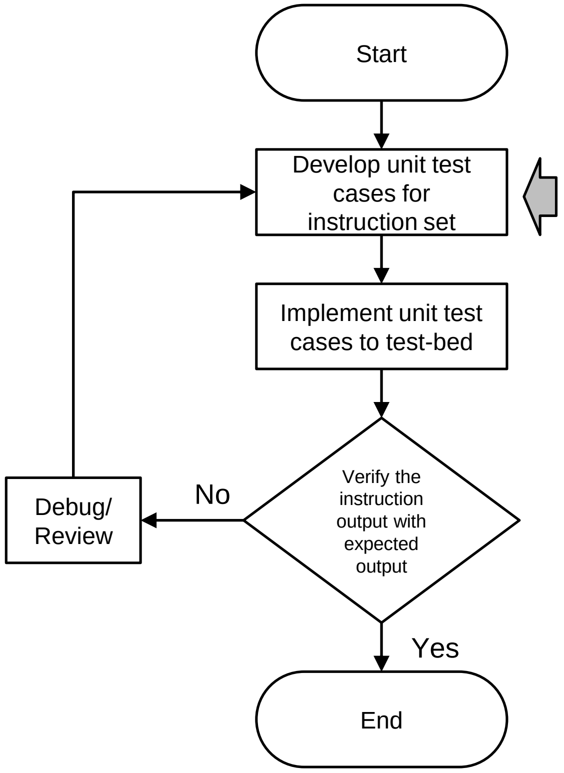


```
void tms_ldi(cpu *m, uint32_t n){
    TwoParaOper1 *c1 = m->code_operand[n];
    uint32_t val_src, val_dst;

    switch (c1->field.G_bin){
        case 0x0: // register addressing mode
        if (c1->field.src_r < 8){ val_src = (uint32_t)m->er[c1-
        >field.src_r].val; }
        else{ val_src = m->oer[c1->field.src_r - 8].val; }
        break;
        case 0x1: // direct addressing mode
        val_src = readmem_int(m, directaddress(m, c1->field.src_disp));
        break;
        case 0x2: // indirect addressing mode
        val_src = readmem_int(m, indirectaddress(m, c1->field.src_mod, c1-
        >field.src_r, c1->field.src_disp));
        break;
        case 0x3: // immediate addressing mode
        val_src = (uint32_t)((int16_t)c1->field.src_imm);
        break;
    }

    if (c1->field.dst_r < 8) {
        m->er[c1->field.dst_r].val = val_src;
        set_flag(m, FLAG_UF, 0);
        set_flag(m, FLAG_V, 0);

        val_dst = m->er[c1->field.dst_r].val;
        if ((int32_t)val_dst < 0) { set_flag(m, FLAG_N, 1); }
        else { set_flag(m, FLAG_N, 0); }
        if (val_dst == 0) { set_flag(m, FLAG_Z, 1); }
        else { set_flag(m, FLAG_Z, 0); }
        }
    else{
         m->oer[c1->field.dst_r - 8].val = val_src;
    }

    m->pc++;
}
```

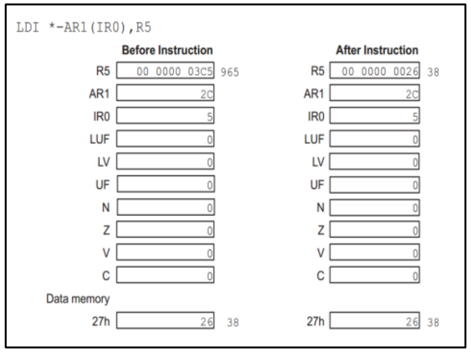
Emulated operation of **LDI (load integer)** instruction set in test-bed

Appendix IV.

- **V&V of NPP safety software test-bed (*NSTM*)**
  - <u>**Instruction set testing**</u> (using *CppUnit**) : A total of 2090 unit test cases were developed and tested to verify the correctness of emulated instruction set.


Start
Develop unit test cases for instruction set
Implement unit test cases to test-bed
Verify the instruction output with expected output
No
Debug/ Review
Yes
End

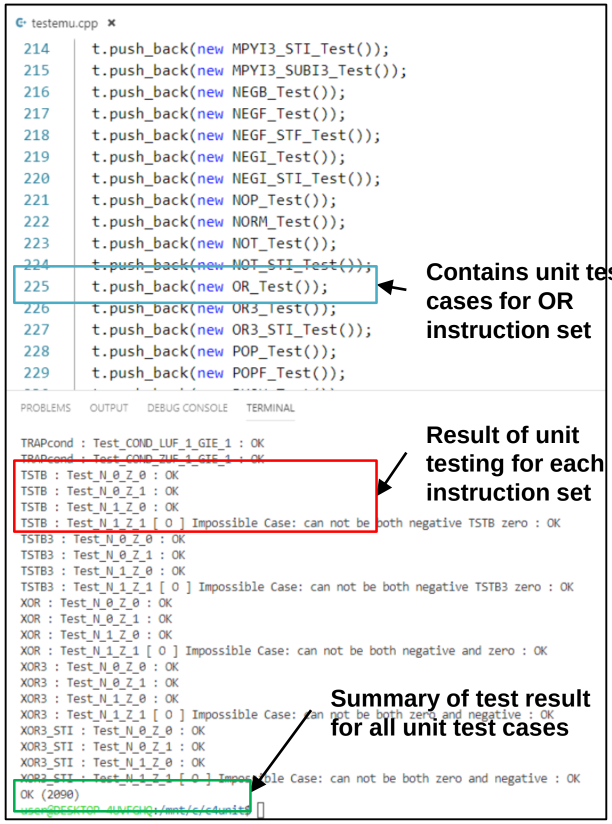
Verification process of the software test-bed with instruction set unit test cases



```
LDI *-AR1(IR0),R5
```

	Before Instruction			After Instruction	
R5	00 0000 03C5	965	R5	00 0000 0026	38
AR1	2C		AR1	2C	
IR0	5		IR0	5	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
27h	26	38	27h	26	38

Example of unit test case developed for **LDI** (load integer) instruction


testemu.cpp
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Contains unit test cases for OR instruction set
Result of unit testing for each instruction set
Summary of test result for all unit test cases

```
t.push_back(new MPYI3_STI_Test());
t.push_back(new MPYI3_SUBI3_Test());
t.push_back(new NEGB_Test());
t.push_back(new NEGF_Test());
t.push_back(new NEGF_STF_Test());
t.push_back(new NEGI_Test());
t.push_back(new NEGI_STI_Test());
t.push_back(new NOP_Test());
t.push_back(new NORM_Test());
t.push_back(new NOT_Test());
t.push_back(new NOT_STI_Test());
t.push_back(new OR_Test());
t.push_back(new OR3_Test());
t.push_back(new OR3_STI_Test());
t.push_back(new POP_Test());
t.push_back(new POPF_Test());
```

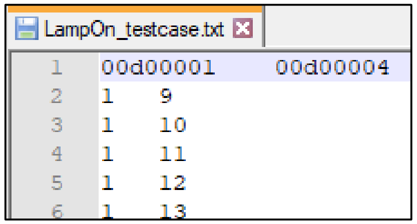
```
TRAPcond : Test_COND_LUF_1_GIE_1 : OK
TRAPcond : Test_COND_ZUF_1_GIE_1 : OK
TSTB : Test_N_0_Z_0 : OK
TSTB : Test_N_0_Z_1 : OK
TSTB : Test_N_1_Z_0 : OK
TSTB : Test_N_1_Z_1 [ O ] Impossible Case: can not be both negative TSTB zero : OK
TSTB3 : Test_N_0_Z_0 : OK
TSTB3 : Test_N_0_Z_1 : OK
TSTB3 : Test_N_1_Z_0 : OK
TSTB3 : Test_N_1_Z_1 [ O ] Impossible Case: can not be both negative TSTB3 zero : OK
XOR : Test_N_0_Z_0 : OK
XOR : Test_N_0_Z_1 : OK
XOR : Test_N_1_Z_0 : OK
XOR : Test_N_1_Z_1 [ O ] Impossible Case: can not be both negative and zero : OK
XOR3 : Test_N_0_Z_0 : OK
XOR3 : Test_N_0_Z_1 : OK
XOR3 : Test_N_1_Z_0 : OK
XOR3 : Test_N_1_Z_1 [ O ] Impossible Case: can not be both zero and negative : OK
XOR3_STI : Test_N_0_Z_0 : OK
XOR3_STI : Test_N_0_Z_1 : OK
XOR3_STI : Test_N_1_Z_0 : OK
XOR3_STI : Test_N_1_Z_1 [ O ] Impossible Case: can not be both zero and negative : OK
OK (2090)
user@DESKTOP-4UVFGHQ:/mnt/c/c4unit$
```

Result of unit testing for software test-bed

* *CppUnit* : xUnit tool for C/C++ programming languages

Appendix IV.

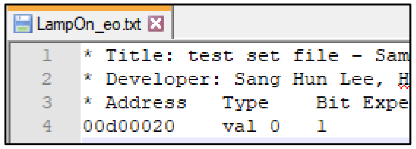
- **V&V of NPP safety software test-bed (*NSTM*)**
  - <u>**Functional testing**</u> : The test cases for benchmark programs were developed and tested to verify the overall functionality of test-bed.
    - Lamp On/Off software : 22 cases for *Lamp On* scenario
    - KNICS IDiPS-RPS BP software : 659 cases for *trip* scenario (15 trip logics)



LampOn_testcase.txt

```
00d00001     00d00004
1    9
1    10
1    11
1    12
1    13
```

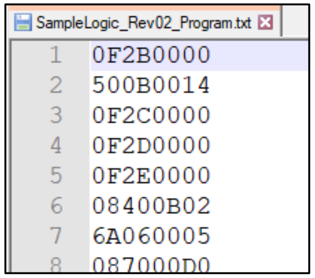
Test case file



LampOn_eo.txt

```
* Title: test set file - Sam
* Developer: Sang Hun Lee, H
* Address    Type    Bit Expe
00d00020    val 0    1
```

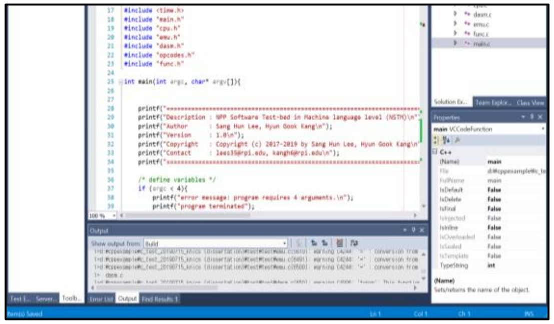
Expected output file



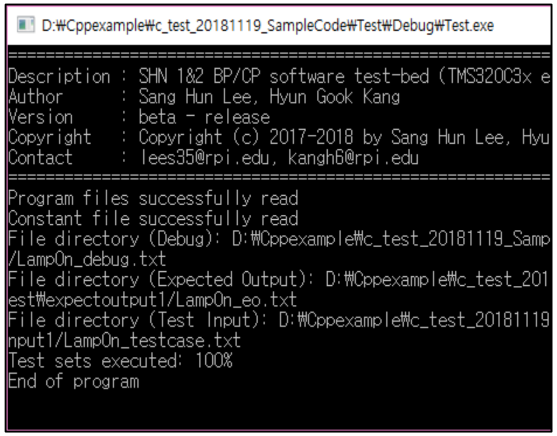
SampleLogic_Rev02_Program.txt

```
0F2B0000
500B0014
0F2C0000
0F2D0000
0F2E0000
08400B02
6A060005
087000D0
```

Program file (executable code)



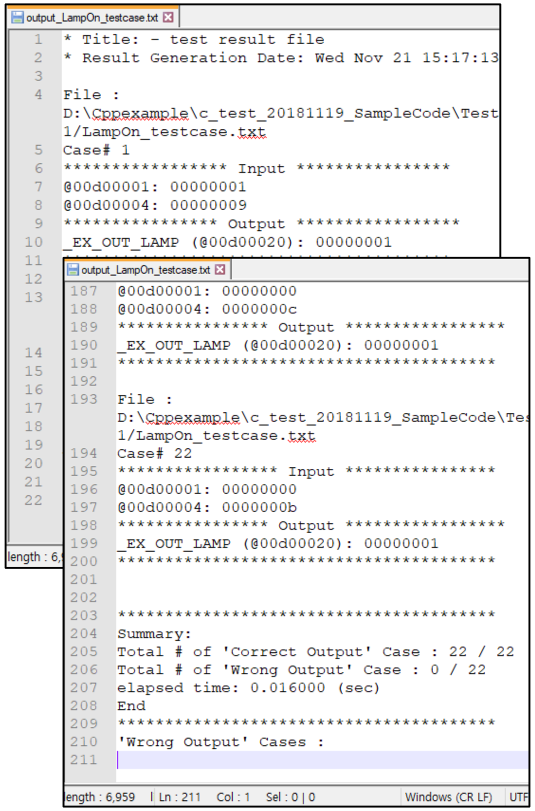
```
#include <time.h>
#include "main.h"
#include "cpu.h"
#include "emu.h"
#include "dasm.h"
#include "opcodes.h"
#include "func.h"

int main(int argc, char* argv[]){


    printf("======================================================================
    printf("Description : NPP Software Test-bed in Machine language level (NSTM)\n"
    printf("Author      : Sang Hun Lee, Hyun Gook Kang\n");
    printf("Version     : 1.0\n");
    printf("Copyright   : Copyright (c) 2017-2019 by Sang Hun Lee, Hyun Gook Kang\n"
    printf("Contact     : lees35@rpi.edu, kangh6@rpi.edu\n");
    printf("======================================================================

    /* define variables */
    if (argc < 4){
        printf("error message: program requires 4 arguments.\n");
        printf("program terminated");
```



D:₩Cppexample₩c_test_20181119_SampleCode₩Test₩Debug₩Test.exe

```
==============================================================================
Description : SHN 1&2 BP/CP software test-bed (TMS320C3x e
Author      : Sang Hun Lee, Hyun Gook Kang
Version     : beta - release
Copyright   : Copyright (c) 2017-2018 by Sang Hun Lee, Hyu
Contact     : lees35@rpi.edu, kangh6@rpi.edu
==============================================================================
Program files successfully read
Constant files successfully read
File directory (Debug): D:₩Cppexample₩c_test_20181119_Samp
/LampOn_debug.txt
File directory (Expected Output): D:₩Cppexample₩c_test_201
est₩expectoutput1/LampOn_eo.txt
File directory (Test Input): D:₩Cppexample₩c_test_20181119
nput1/LampOn_testcase.txt
Test sets executed: 100%
End of program
```

Screenshot of test-bed execution



output_LampOn_testcase.txt

```
* Title: - test result file
* Result Generation Date: Wed Nov 21 15:17:13

File :
D:\Cppexample\c_test_20181119_SampleCode\Test
1/LampOn_testcase.txt
Case# 1
**************** Input ****************
@00d00001: 00000001
@00d00004: 00000009
*************** Output ****************
_EX_OUT_LAMP (@00d00020): 00000001
```

output_LampOn_testcase.txt

```
@00d00001: 00000000
@00d00004: 0000000c
**************** Output ****************
_EX_OUT_LAMP (@00d00020): 00000001
***************************************


File :
D:\Cppexample\c_test_20181119_SampleCode\Tes
1/LampOn_testcase.txt
Case# 22
**************** Input ****************
@00d00001: 00000000
@00d00004: 0000000b
**************** Output ****************
_EX_OUT_LAMP (@00d00020): 00000001
***************************************



***************************************
Summary:
Total # of 'Correct Output' Case : 22 / 22
Total # of 'Wrong Output' Case : 0 / 22
elapsed time: 0.016000 (sec)
End
***************************************
'Wrong Output' Cases :
```

length : 6,959   Ln : 211   Col : 1   Sel : 0 | 0   Windows (CR LF)   UTF

Test result file

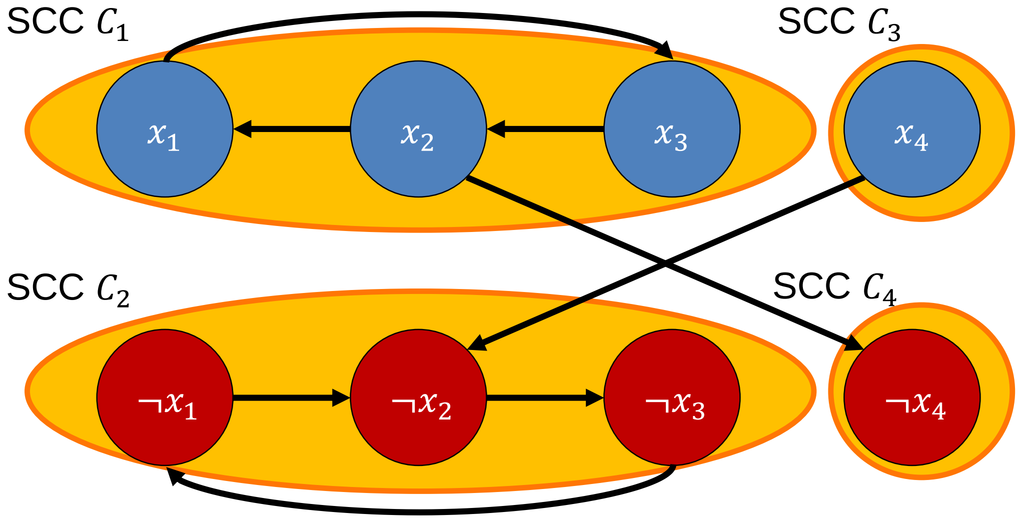
Appendix V.

- Theoretical Basis of Exhaustive Test Case Generation
  - Example algorithms to solve an SAT example (SSC):

$$f_1 = x_1 \lor x_2 \equiv \neg x_1 \Rightarrow x_2$$: *Satisfiable*

$$f_2 = (x_1 \lor x_1) \land (\neg x_1 \lor \neg x_1) \equiv \neg x_1 \Rightarrow x_1,\ x_1 \Rightarrow \neg x_1$$: *Unsatisfiable*

$$f_3 = (x_1 \lor \neg x_2) \land (x_2 \lor \neg x_3) \land (x_3 \lor \neg x_1) \land (\neg x_4 \lor \neg x_2)$$
$$\equiv \neg x_1 \Rightarrow \neg x_2,\ \neg x_2 \Rightarrow \neg x_3,\ \neg x_3 \Rightarrow \neg x_1,\ x_4 \Rightarrow \neg x_2$$



Check satisfiability:
If x_i and $\neg x_i$ are at same SCC, *unsat*
else, *sat*
$\rightarrow$ in f_3 case, *sat*

Derive interpretation:
For each SCC C_1, set vertices' value to false,
(to avoid *true* $\rightarrow$ *false* cases),
and check the contradictions between SCCs.

$\rightarrow$ in f_3 case,
$\{x_1, x_2, x_3, x_4\} = \{false, false, false, true\}$

* SCC : Strongly Connected Component (A directed graph is strongly connected if there is a path between all pairs of vertices)

Appendix V.

- Theoretical Basis of Exhaustive Test Case Generation
  - Expansion from SAT to SMT problem:

$$f_4 = \overbrace{(x + 5}^{\text{arithmetic}} = read(A,5)) \wedge \big(A \neq B \vee \overbrace{read(B,5)}^{\text{Arrays}} \leq \overbrace{g(x)}^{\text{Uninterpreted functions}}\big)$$

$\Downarrow$ Abstract to Propositional Logic

$$f_4 = (\quad P \quad) \wedge (\quad Q \quad \vee \quad R \quad)$$

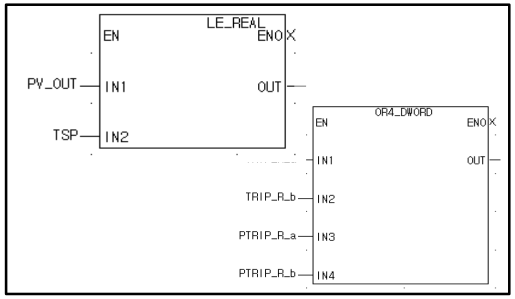
$\Downarrow$ Find satisfying assignment

$x + 5 = read(A,5)$, $A \neq B$ $\Rightarrow$ Determine if consistent according to theory solver (arithmetic, equalities, etc.)

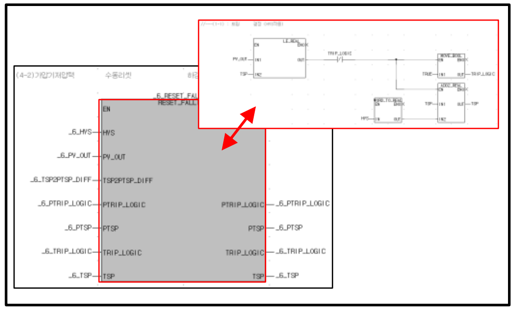
* read(a, i) means to read the i-th element in array a.

Appendix VI.

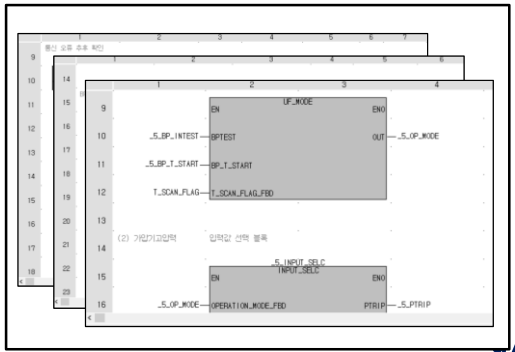
- NPP safety software language – FBD/LD program
  - FBD is a graphical language used for PLC design that describes the function between input and output variables expressed with a set of elementary blocks.
    - Definition of Function Block (*FB*)
      - FB = <Name, IP, OP, BD>
        - $f_{FB}: I_{FB} \rightarrow O_{FB}$
          - » Name : a name of function block,
          - » IP: a set of input ports, $I_{FB} = I_1 * \cdots * I_n$
          - » OP: a set of output ports, $O_{FB} = O_o$
          - » BD: behavioral description of function block.



    - Definition of Component FBD (*Comp_FBD*)
      - Component_FBD = <FBs, T, I, O>,
        - $f_{Comp\_FBD}: I_{Comp\_FBD} \rightarrow O_{Comp\_FBD}$
          - » FBs : a set of FBs,
          - » T: A set of transition $(FB_i.OP_m, FB_j.IP_n)$ btw FBs $\forall(FB_i.OP_m, FB_j.IP_n) \in T$
          - » I: a set of FB.IP not included in T, $I_{Comp\_FBD} = I_1 * \cdots * I_m$
          - » O: a set of FB.OP not included in T, $O_{Comp\_FBD} = O_1 * \cdots * O_m$



    - Definition of System FBD (*Sys_FBD*)
      - System_FBD = <FBDs, T, I, O>,
        - $f_{Sys\_FBD}: I_{Sys\_FBD} \rightarrow O_{Sys\_FBD}$
          - » FBDs : a set of *Component_FBDs*,
          - » T: a set of transition $(FBD_i.O_m, FBD_j.I_n)$ btw *Component_FBDs*, $\forall(FBD_i.O_m), FBD_j.I_n) \in T$
          - » I: a set of FBD.I not included in T, $I_{Sys\_FBD} = I_{S1} * \cdots * I_{Sm}$
          - » O: a set of FBD.O not included in T, $O_{Sys\_FBD} = O_{S1} * \cdots * O_{Sn}$



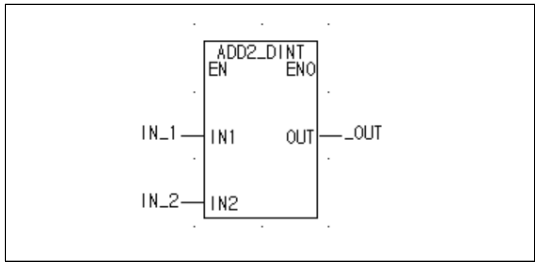
Appendix VII.

Translation from FBD into SMT format

- The FBD-to-SMT translation rules are developed based on FBD formal definition.
- The translation starts with generating SMT formulas for all FBs, and continues for component FBD and system FBD.

Function Block

(*FB = <Name, IP, OP, BD>*)



$FB_{ADD2\_DINT} = < Name, IP, OP, BD >$

Name = ADD2_DINT

IP = {EN, IN1, IN2}

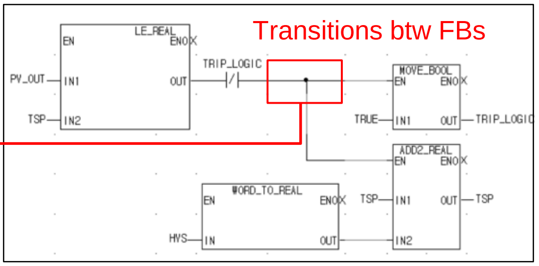
OP = {ENO, OUT}

BD = IN1 + IN2

```
def ADD2_DINT(EN, IN1, IN2):
    # -- define BD of FB --
    BD = IN1 + IN2
    # -- define ENO/outputs of FB --
    ENO1 = If(EN, True, False)
    OUT1 = If(EN, BD, OUT)
    # -- return ENO/outputs of FB --
    return ENO1, OUT1
```

Component FBDs

(*Comp_FBD = <FBs, T_{comp}, I, O>*)



$RESET\_FALLING = < FBs, T_{comp}, I, O >$

FBs = {LE_REAL, MOVE_BOOL, …}

T_{comp} = {MOVE_BOOL.in_EN = LE_REAL.OUT∧ ¬(TRIP_LOGIC), …}

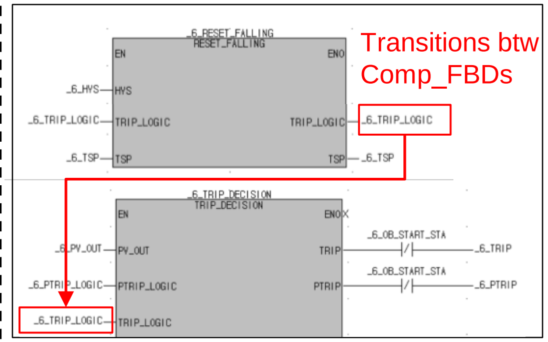
I = {EN, PV_OUT, TSP, HYS}

O = {ENO, TRIP_LOGIC, TSP}

```
def RESET_FALLING(EN,PV_OUT,in_HYS,
  in_PV_OUT,…,ENO,…,out_TRIP_LOGIC)
  # -- temporary variables --
  temp1 = False
  # -- function operation --
  LE_REAL0_OUT = LE_REAL(True, in_PV_OUT
                              , in_TSP, …)
  temp1 = And(LE_REAL0_OUT,
              Not(in_TRIP_LOGIC))
  TRIP_LOGIC1 = MOVE_BOOL(temp1, True, …)
  # return ENO/outputs of FB --
  ENO1 = If(EN, True, False)
  TRIP_LOGIC5 = If(EN, TRIP_LOGIC4
                  , out_TRIP_LOGIC)
  return ENO1, …, TRIP_LOGIC5
```

System FBDs

(*Sys_FBD = <Comp_FBDs, T_{sys}, I, O>*)



$BP\_k\_Trip\_LogicA = < Comp\_FBDs, T_{sys}, I, O >$

Comp_FBDs = {_6_RESET_FALLING, _6_TRIP_DECISION, …}

T_{sys} = {_6_TRIP_DECISION.in_TRIP_LOGIC = _6_RESET_FALLING.out_TRIP_LOGIC, …}

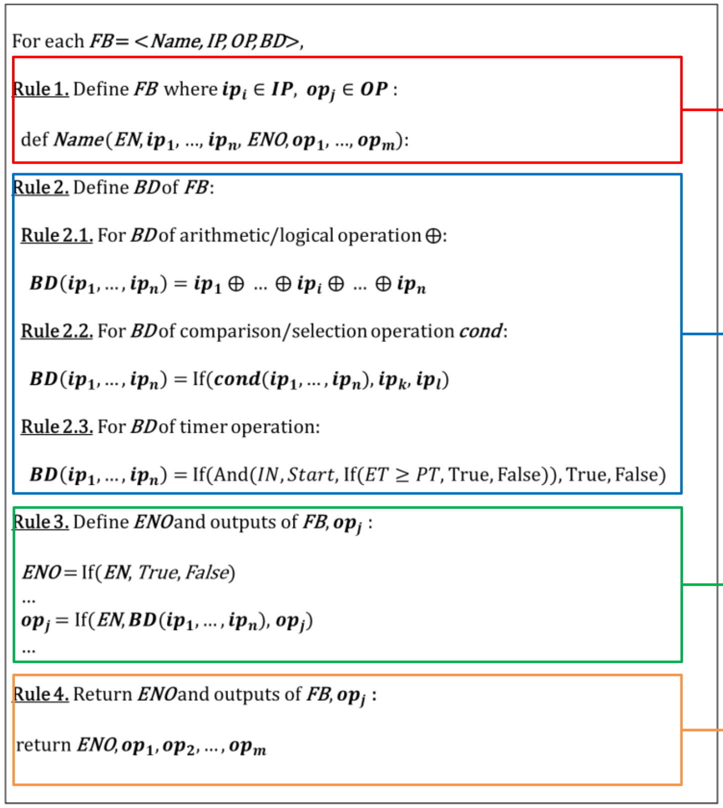
I = {_6_HYS, _6_TRIP_LOGIC, …}

O = {_6_TRIP, _6_TSP, …}

```
#define FB
def BP_k_Trip_LogicA(BP_k_Trip_LogicA_in):
    # define ENO and outputs of FB
    …, _6_TRIP_LOGIC2, … = RESET_FALLING(
        …, _6_TRIP_LOGIC1, …)
    …, TRIP_DECISION51_out_TRIP, … =
        TRIP_DECISION(…, _6_TRIP_LOGIC2, …)
    …
    # return ENO and outputs of FB
    return BP_k_Trip_LogicA_out
```

Appendix VII.

- Translation of Function Block (FB) to SMT formula
  - **Rule 1** : Define the elements of FB (name, I/O ports)
  - **Rule 2** : Define the FB operation
  - **Rule 3** : Define the FB output ports
  - **Rule 4** : Return the FB output ports



For each *FB*= <*Name, IP, OP, BD*>,

Rule 1. Define *FB* where $\boldsymbol{ip_i} \in \boldsymbol{IP},\ \boldsymbol{op_j} \in \boldsymbol{OP}$:

def $Name(EN, \boldsymbol{ip_1}, \ldots, \boldsymbol{ip_n}, ENO, \boldsymbol{op_1}, \ldots, \boldsymbol{op_m})$:

Rule 2. Define *BD* of *FB*:

Rule 2.1. For *BD* of arithmetic/logical operation $\oplus$:

$\boldsymbol{BD}(\boldsymbol{ip_1}, \ldots, \boldsymbol{ip_n}) = \boldsymbol{ip_1} \oplus \ldots \oplus \boldsymbol{ip_i} \oplus \ldots \oplus \boldsymbol{ip_n}$

Rule 2.2. For *BD* of comparison/selection operation *cond*:

$\boldsymbol{BD}(\boldsymbol{ip_1}, \ldots, \boldsymbol{ip_n}) = \mathrm{If}(\boldsymbol{cond}(\boldsymbol{ip_1}, \ldots, \boldsymbol{ip_n}), \boldsymbol{ip_k}, \boldsymbol{ip_l})$

Rule 2.3. For *BD* of timer operation:

$\boldsymbol{BD}(\boldsymbol{ip_1}, \ldots, \boldsymbol{ip_n}) = \mathrm{If}(\mathrm{And}(IN, Start, \mathrm{If}(ET \geq PT, \mathrm{True}, \mathrm{False})), \mathrm{True}, \mathrm{False})$

Rule 3. Define *ENO* and outputs of *FB*, $\boldsymbol{op_j}$:

$ENO = \mathrm{If}(EN, True, False)$

...

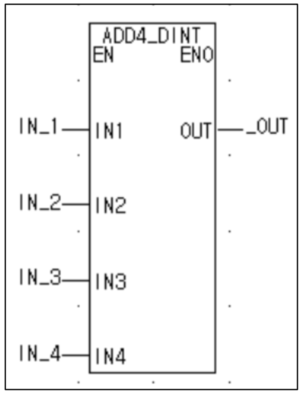
$\boldsymbol{op_j} = \mathrm{If}(EN, \boldsymbol{BD}(\boldsymbol{ip_1}, \ldots, \boldsymbol{ip_n}), \boldsymbol{op_j})$

...

Rule 4. Return *ENO* and outputs of *FB*, $\boldsymbol{op_j}$:

return $ENO, \boldsymbol{op_1}, \boldsymbol{op_2}, \ldots, \boldsymbol{op_m}$

FBD-to-SMT translation rules for FBs



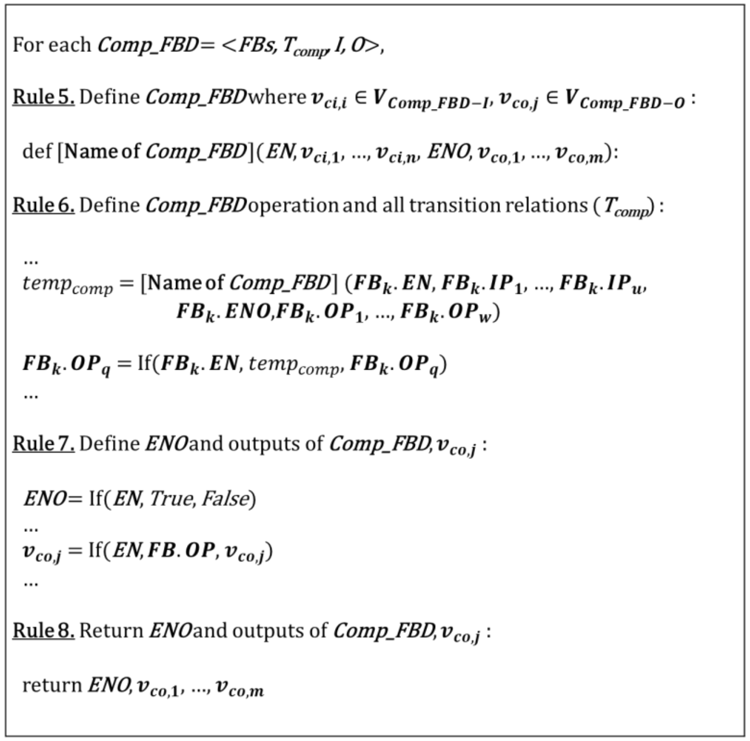
$$FB_{ADD4\_DINT} =< Name, IP, OP, BD >$$
$$Name = ADD4\_DINT$$
$$IP = \{EN, IN1, IN2, IN3, IN4\}$$
$$\mathrm{OP} = \{ENO, OUT\}$$
$$BD = IN1 + IN2 + IN3 + IN4$$

```
#define FB
def ADD4_DINT(EN, IN1, IN2, IN3, IN4, ENO, OUT):

    # define BD of FB
    BD = IN1 + IN2 + IN3 + IN4

    # define ENO and outputs of FB
    ENO2 = If(EN, True, False)
    OUT2 = If(EN, BD, OUT)

    # return ENO and outputs of FB
    return ENO2, OUT2
```

Formal definition and translation for example FB, *ADD4_DINT*

Appendix VII.

- Translation of Component FBD (*Comp_FBD*) to SMT formula
  - **Rule 5** : Define the elements of component FBD (name, I/O ports)
  - **Rule 6** : Define the program flow according to execution order of FBs in *Comp_FBD*
  - **Rule 7** : Define the *Comp_FBD* output ports
  - **Rule 8** : Return the *Comp_FBD* output ports



For each *Comp_FBD*= <*FBs*, T_{comp}, *I*, *O*>,

Rule 5. Define *Comp_FBD* where $v_{ci,i} \in V_{Comp\_FBD-I}$, $v_{co,j} \in V_{Comp\_FBD-O}$:

def [Name of *Comp_FBD*]($EN, v_{ci,1}, \dots, v_{ci,n}, ENO, v_{co,1}, \dots, v_{co,m}$):

Rule 6. Define *Comp_FBD* operation and all transition relations (T_{comp}) :

…

$temp_{comp}$ = [Name of *Comp_FBD*] ($FB_k.EN, FB_k.IP_1, \dots, FB_k.IP_u,$
$FB_k.ENO, FB_k.OP_1, \dots, FB_k.OP_w$)

$FB_k.OP_q = \mathrm{If}(FB_k.EN, temp_{comp}, FB_k.OP_q)$

…

Rule 7. Define *ENO* and outputs of *Comp_FBD*, $v_{co,j}$:

$ENO = \mathrm{If}(EN, True, False)$

…

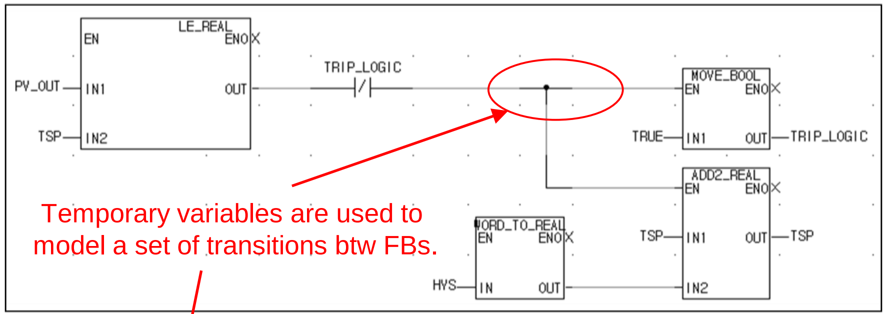
$v_{co,j} = \mathrm{If}(EN, FB.OP, v_{co,j})$

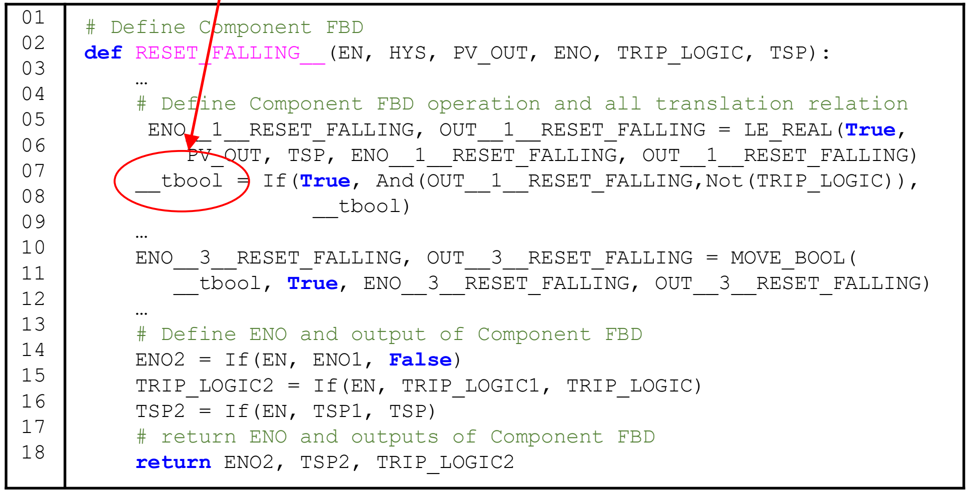
…

Rule 8. Return *ENO* and outputs of *Comp_FBD*, $v_{co,j}$:

return $ENO, v_{co,1}, \dots, v_{co,m}$

FBD-to-SMT translation rules for component FBDs



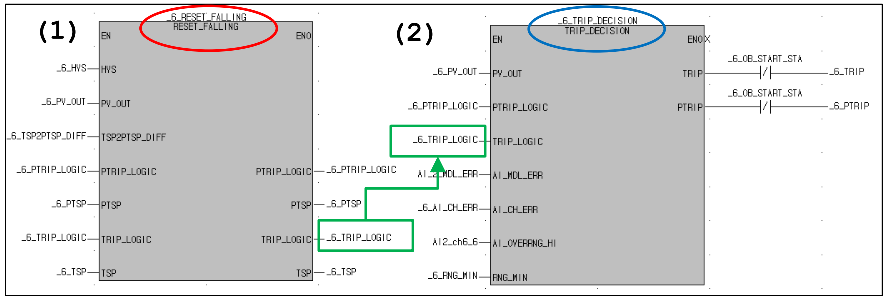


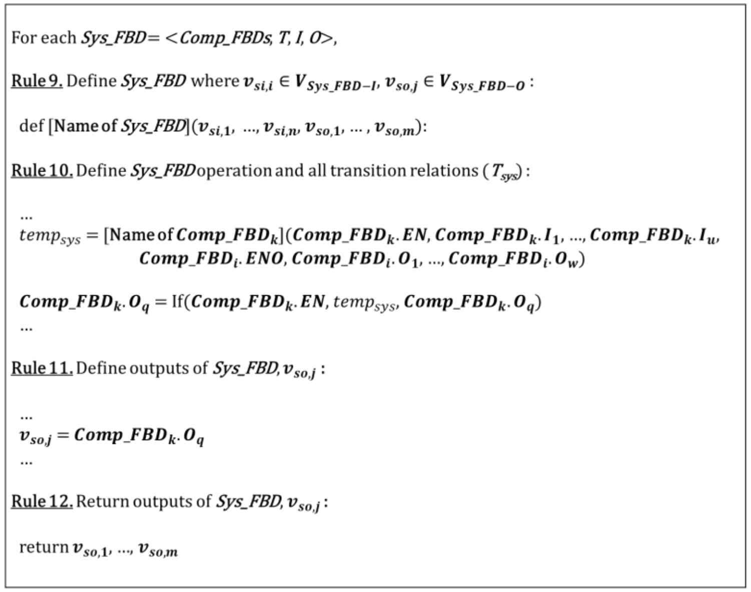
```
# Define Component FBD
def RESET_FALLING__(EN, HYS, PV_OUT, ENO, TRIP_LOGIC, TSP):
    …
    # Define Component FBD operation and all translation relation
     ENO__1__RESET_FALLING, OUT__1__RESET_FALLING = LE_REAL(True,
        PV_OUT, TSP, ENO__1__RESET_FALLING, OUT__1__RESET_FALLING)
    __tbool = If(True, And(OUT__1__RESET_FALLING,Not(TRIP_LOGIC)),
              __tbool)
    …
    ENO__3__RESET_FALLING, OUT__3__RESET_FALLING = MOVE_BOOL(
       __tbool, True, ENO__3__RESET_FALLING, OUT__3__RESET_FALLING)
    …
    # Define ENO and output of Component FBD
    ENO2 = If(EN, ENO1, False)
    TRIP_LOGIC2 = If(EN, TRIP_LOGIC1, TRIP_LOGIC)
    TSP2 = If(EN, TSP1, TSP)
    # return ENO and outputs of Component FBD
    return ENO2, TSP2, TRIP_LOGIC2
```

An example of translation for the component FBDs

Appendix VII.

- Translation of System FBD (*Sys_FBD*) to SMT formula
  - **<u>Rule 9</u>** : Define the elements of system FBD (name, I/O ports)
  - **<u>Rule 10</u>** : Define the program flow according to execution order in *Sys_FBD*
  - **<u>Rule 11</u>** : Define the *Sys_FBD* output ports
  - **<u>Rule 12</u>** : Return the *Sys_FBD* output ports





For each *Sys_FBD*= <*Comp_FBDs, T, I, O*>,

<u>Rule 9.</u> Define *Sys_FBD* where $v_{si,i} \in V_{Sys\_FBD-I}$, $v_{so,j} \in V_{Sys\_FBD-O}$:

def [**Name of** *Sys_FBD*]($v_{si,1}$, …, $v_{si,n}$, $v_{so,1}$, … , $v_{so,m}$):

<u>Rule 10.</u> Define *Sys_FBD* operation and all transition relations (T_{sys}) :

…

$temp_{sys}$ = [**Name of** $\boldsymbol{Comp\_FBD_k}$]($\boldsymbol{Comp\_FBD_k.EN}$, $\boldsymbol{Comp\_FBD_k.I_1}$, …, $\boldsymbol{Comp\_FBD_k.I_u}$, $\boldsymbol{Comp\_FBD_i.ENO}$, $\boldsymbol{Comp\_FBD_i.O_1}$, …, $\boldsymbol{Comp\_FBD_i.O_w}$)

$\boldsymbol{Comp\_FBD_k.O_q}$ = If($\boldsymbol{Comp\_FBD_k.EN}$, $temp_{sys}$, $\boldsymbol{Comp\_FBD_k.O_q}$)

…

<u>Rule 11.</u> Define outputs of *Sys_FBD*, $v_{so,j}$:

…

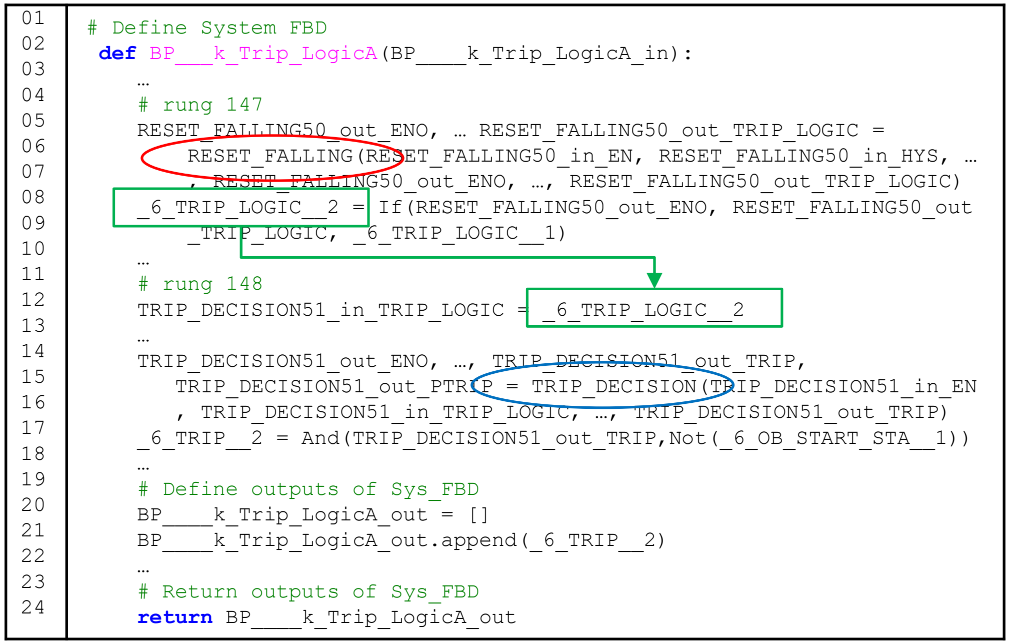
$v_{so,j} = \boldsymbol{Comp\_FBD_k.O_q}$

…

<u>Rule 12.</u> Return outputs of *Sys_FBD*, $v_{so,j}$:

return $v_{so,1}$, …, $v_{so,m}$

FBD-to-SMT translation rules for system FBD



```
# Define System FBD
 def BP___k_Trip_LogicA(BP____k_Trip_LogicA_in):
    …
    # rung 147
    RESET_FALLING50_out_ENO, … RESET_FALLING50_out_TRIP_LOGIC =
        RESET_FALLING(RESET_FALLING50_in_EN, RESET_FALLING50_in_HYS, …
        , RESET_FALLING50_out_ENO, …, RESET_FALLING50_out_TRIP_LOGIC)
    _6_TRIP_LOGIC__2 = If(RESET_FALLING50_out_ENO, RESET_FALLING50_out
        _TRIP_LOGIC, _6_TRIP_LOGIC__1)
    …
    # rung 148
    TRIP_DECISION51_in_TRIP_LOGIC = _6_TRIP_LOGIC__2
    …
    TRIP_DECISION51_out_ENO, …, TRIP_DECISION51_out_TRIP,
       TRIP_DECISION51_out_PTRIP = TRIP_DECISION(TRIP_DECISION51_in_EN
       , TRIP_DECISION51_in_TRIP_LOGIC, …, TRIP_DECISION51_out_TRIP)
    _6_TRIP__2 = And(TRIP_DECISION51_out_TRIP,Not(_6_OB_START_STA__1))
    …
    # Define outputs of Sys_FBD
    BP____k_Trip_LogicA_out = []
    BP____k_Trip_LogicA_out.append(_6_TRIP__2)
    …
    # Return outputs of Sys_FBD
    return BP____k_Trip_LogicA_out
```

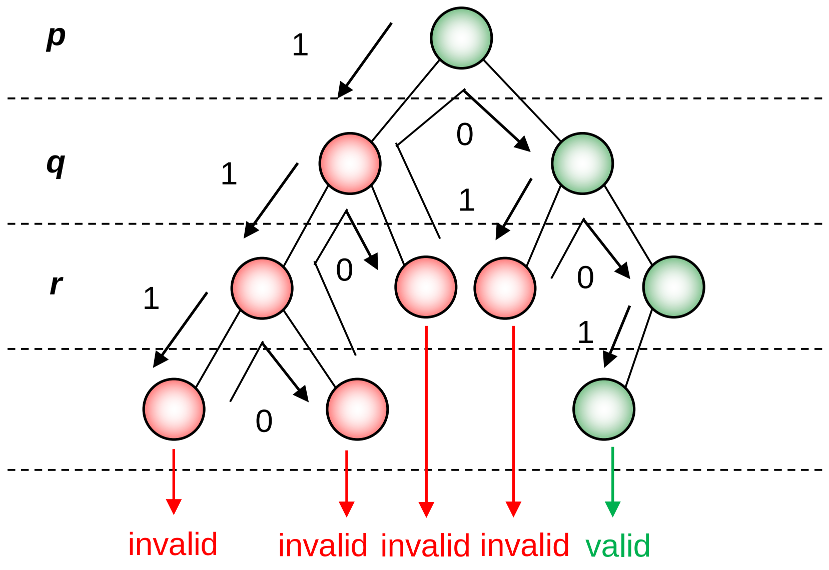
An example of translation for the system FBD

Appendix VIII.

- Background on SMT Solving Techniques – ***DPLL*** algorithm
  - DPLL algorithm is a ***complete***, ***backtracking***-based search algorithm for solving the CNF-SAT problem.
    - *Complete*: guarantees to find a solution if there is any
    - *Backtracking*: exercise all possible paths to solve SAT/SMT problem.

$$F = \neg p \wedge \neg q \wedge r$$

$$\{p = false, q = false, r = true\}$$



- Existing SMT solver: CVC4 (Stanford), Yices (SRI), **Z3 (Microsoft)** [7]
  - Z3 ~ *DPLL-(T)*, a theorem prover for first-order logic about an arbitrary theory *T*.

* DPLL: Davis–Putnam–Logemann–Loveland

Appendix VIII.

- SMT Solving Technique – DPLL algorithm ($DPLL(F,U)$)
  - UP(F, U); - Unit-propagate
  - if F contains the empty clause ($F = \bot$) then return;
  - if F is empty formula ($F = \top$), then exit with model of U;
  - $L \leftarrow$ a literal containing an atom from F;
  - $DPLL(F|_L, U \cup \{L\})$;
  - $DPLL(F|_{\bar{L}}, U \cup \{\bar{L}\})$;

$$\boldsymbol{F = (\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r)}$$

$$F = (\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r)$$

$$\begin{tikzcd}
 & \bullet \arrow[dl, "p"'] \arrow[dr, "\neg p"] & & \\
\bot & & (q \vee r) \wedge (\neg q \vee \neg r) \arrow[dl, "q"'] \arrow[dr, "\neg q"] & \\
 & \neg r & & r
\end{tikzcd}$$

An example of SMT solving using DPLL algorithm

```
$DP((\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r),\ \emptyset)$
└ $UP((\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r),\ \emptyset)$
  $L \leftarrow \neg p$
  $DP\big((\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r)|_{\neg p}, \{\neg p\}\big)$
  $DP\big((q \vee r) \wedge (\neg q \vee \neg r), \{\neg p\}\big)$
  └ $UP\big((q \vee r) \wedge (\neg q \vee \neg r), \{\neg p\}\big)$
    $L \leftarrow q$
    $DP\big((q \vee r) \wedge (\neg q \vee \neg r)|_q, \{\neg p, q\}\big)$
    └ $UP(\neg r, \{\neg p, q\})$
      $L \leftarrow \neg r$
      $DP(\top, \{\neg p, q, \neg r\})$
      └ return model $\{\neg p, q, \neg r\}$
```

Appendix VIII.

- Z3: DPLL-based SAT + Theory solver (Arithmatic, Array, Bit-vector)

$$\Phi = (x+1>0 \ \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg(x+y>0)$$

- Invoke DPLL(T) for theory T = LIA (Linear Integer Arithmetic)
  - Map : $\{A \leftrightarrow x+1>0,\ B \leftrightarrow x+y>0,\ C \leftrightarrow x<0,\ D \leftrightarrow x+y>4\}$
  - Invoke SAT solver:
    - Propagate: $B \rightarrow false$, Propagate: $A \rightarrow true$
    - Decide: $C \rightarrow true$

$$\Phi = (\ A \ \vee \ B\ ) \wedge (\ C \ \vee \ D\ ) \wedge \neg B$$

  - Invoke theory solver for LIA on: $\{A, \neg B, C\} \rightarrow \{x+1>0,\ \neg(x+y>0),\ x<0\}$
    - $x+1>0 \wedge x<0$ is LIA-unsatisfiable.
    - $\rightarrow (\neg A \vee \neg C)$ is added to list of clauses.

$$\Phi = (\ A \ \vee \ B\ ) \wedge (\ C \ \vee \ D\ ) \wedge \neg B \wedge (\ \neg A \ \vee \ \neg C\ )$$

Appendix VIII.

- Z3: DPLL-based SAT + Theory solver (Arithmatic, Array, Bit-vector)

$$\Phi = (x + 1 > 0 \ \vee x + y > 0) \wedge (x < 0 \vee x + y > 4) \wedge \neg(x + y > 0)$$

- Invoke DPLL(T) for theory T = LIA (Linear Integer Arithmetic)
  - Invoke SAT solver:
    - Backtrack decision on C ($C \rightarrow true$)
    - Propagate $C \rightarrow false$
    - Propagate $D \rightarrow true$

$$\Phi = (\ A \ \vee \ B \ ) \wedge (\ C \ \vee \ D \ ) \wedge \neg B \wedge (\ \neg A \ \vee \ \neg C \ )$$

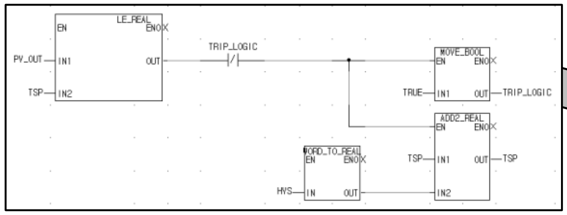
  - Invoke LIA on: $\{A, \neg B, \neg C, D\} \rightarrow \{x + 1 > 0,\ \neg(x + y > 0),\ x < 0,\ \ x + y > 4\}$
    - $\neg(x + y > 0) \wedge x + y > 4$ is LIA-unsatisfiable.
    - $\rightarrow (B \vee \neg D)$ is added to list of clauses.

$$\Phi = (\ A \ \vee \ B \ ) \wedge (\ C \ \vee \ D \ ) \wedge \neg B \wedge (\neg A \vee \neg C) \wedge (B \ \vee \ \neg D \ )$$

  - No decisions to backtrack → The formula is LIA-unsatisfiable.

Appendix IX.

- Single Test Case Generation for NPP Safety Software
  - Algorithm for a single test case generation of an example FBD program



An example FBD program

STEP 1: Define problem set

1-1) <u>Declaration of the FBD variables</u> (line 6~20)

1-2) <u>Declaration of FBD program logic</u> (line 22~39)

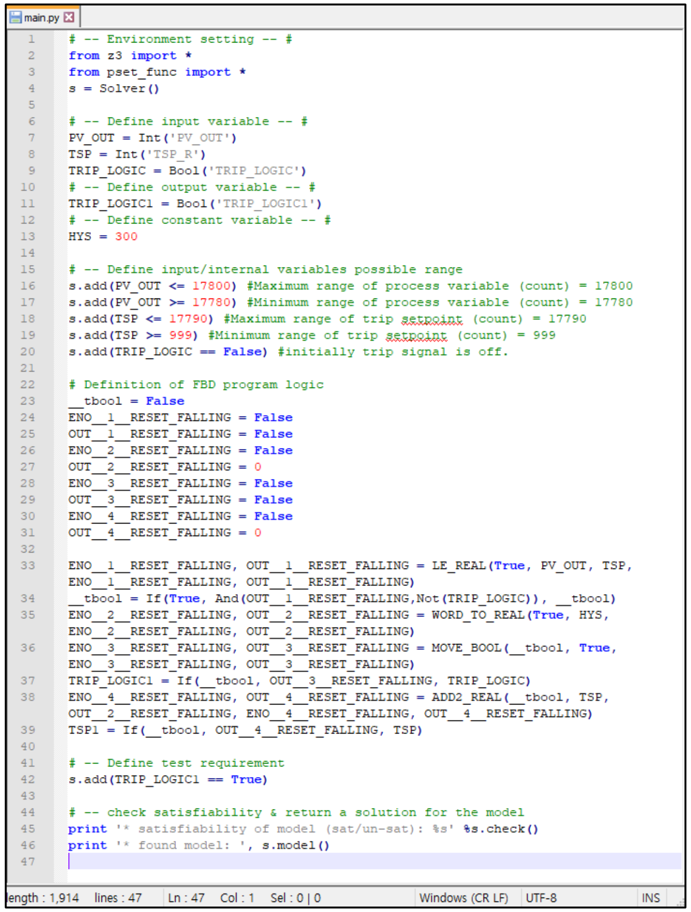
STEP 2: Define test requirement

2-1) <u>Assertion of test requirement</u> (line 41~42):

STEP 3: Solve test requirement

3-1) <u>Check satisfiability given constaint</u> (line 45):

3-2) <u>Find a model for FBD program</u> (line 46):



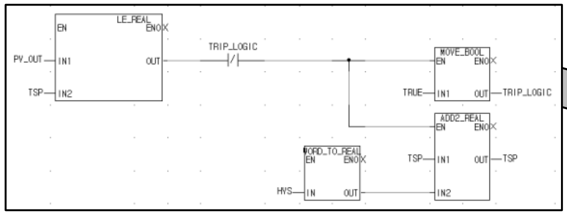
```
 1  # -- Environment setting -- #
 2  from z3 import *
 3  from pset_func import *
 4  s = Solver()
 5
 6  # -- Define input variable -- #
 7  PV_OUT = Int('PV_OUT')
 8  TSP = Int('TSP_R')
 9  TRIP_LOGIC = Bool('TRIP_LOGIC')
10  # -- Define output variable -- #
11  TRIP_LOGIC1 = Bool('TRIP_LOGIC1')
12  # -- Define constant variable -- #
13  HYS = 300
14
15  # -- Define input/internal variables possible range
16  s.add(PV_OUT <= 17800) #Maximum range of process variable (count) = 17800
17  s.add(PV_OUT >= 17780) #Minimum range of process variable (count) = 17780
18  s.add(TSP <= 17790) #Maximum range of trip setpoint (count) = 17790
19  s.add(TSP >= 999) #Minimum range of trip setpoint (count) = 999
20  s.add(TRIP_LOGIC == False) #initially trip signal is off.
21
22  # Definition of FBD program logic
23  __tbool = False
24  ENO__1__RESET_FALLING = False
25  OUT__1__RESET_FALLING = False
26  ENO__2__RESET_FALLING = False
27  OUT__2__RESET_FALLING = 0
28  ENO__3__RESET_FALLING = False
29  OUT__3__RESET_FALLING = False
30  ENO__4__RESET_FALLING = False
31  OUT__4__RESET_FALLING = 0
32
33  ENO__1__RESET_FALLING, OUT__1__RESET_FALLING = LE_REAL(True, PV_OUT, TSP,
    ENO__1__RESET_FALLING, OUT__1__RESET_FALLING)
34  __tbool = If(True, And(OUT__1__RESET_FALLING,Not(TRIP_LOGIC)), __tbool)
35  ENO__2__RESET_FALLING, OUT__2__RESET_FALLING = WORD_TO_REAL(True, HYS,
    ENO__2__RESET_FALLING, OUT__2__RESET_FALLING)
36  ENO__3__RESET_FALLING, OUT__3__RESET_FALLING = MOVE_BOOL(__tbool, True,
    ENO__3__RESET_FALLING, OUT__3__RESET_FALLING)
37  TRIP_LOGIC1 = If(__tbool, OUT__3__RESET_FALLING, TRIP_LOGIC)
38  ENO__4__RESET_FALLING, OUT__4__RESET_FALLING = ADD2_REAL(__tbool, TSP,
    OUT__2__RESET_FALLING, ENO__4__RESET_FALLING, OUT__4__RESET_FALLING)
39  TSP1 = If(__tbool, OUT__4__RESET_FALLING, TSP)
40
41  # -- Define test requirement
42  s.add(TRIP_LOGIC1 == True)
43
44  # -- check satisfiability & return a solution for the model
45  print '* satisfiability of model (sat/un-sat): %s' %s.check()
46  print '* found model: ', s.model()
47
```

```
length : 1,914    lines : 47    Ln : 47    Col : 1    Sel : 0 | 0    Windows (CR LF)    UTF-8    INS
```

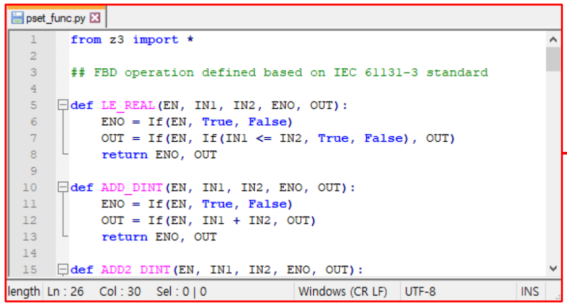
Z3 input file for one test case generation of FBD program

Appendix IX.

- Single Test Case Generation for NPP Safety Software
  - Execution of a single test case generation algorithm for example FBD program



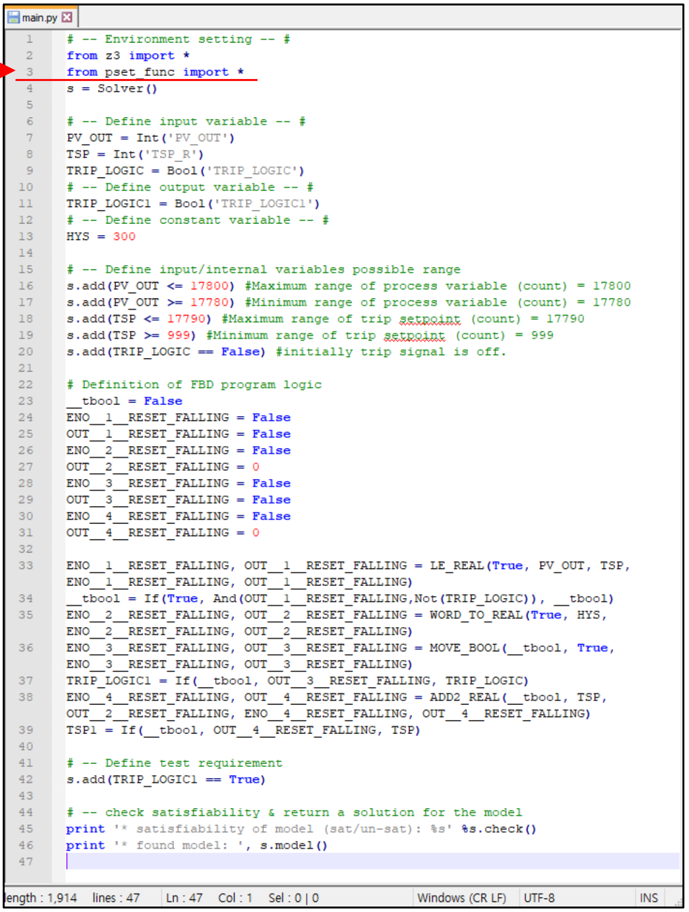
An example FBD program



```
from z3 import *

## FBD operation defined based on IEC 61131-3 standard

def LE_REAL(EN, IN1, IN2, ENO, OUT):
    ENO = If(EN, True, False)
    OUT = If(EN, If(IN1 <= IN2, True, False), OUT)
    return ENO, OUT

def ADD_DINT(EN, IN1, IN2, ENO, OUT):
    ENO = If(EN, True, False)
    OUT = If(EN, IN1 + IN2, OUT)
    return ENO, OUT

def ADD2_DINT(EN, IN1, IN2, ENO, OUT):
```

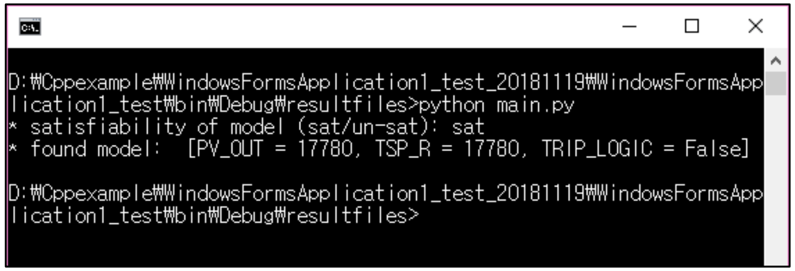
length Ln : 26 Col : 30 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Definition of function blocks



```
# -- Environment setting -- #
from z3 import *
from pset_func import *
s = Solver()

# -- Define input variable -- #
PV_OUT = Int('PV_OUT')
TSP = Int('TSP_R')
TRIP_LOGIC = Bool('TRIP_LOGIC')
# -- Define output variable -- #
TRIP_LOGIC1 = Bool('TRIP_LOGIC1')
# -- Define constant variable -- #
HYS = 300

# -- Define input/internal variables possible range
s.add(PV_OUT <= 17800) #Maximum range of process variable (count) = 17800
s.add(PV_OUT >= 17780) #Minimum range of process variable (count) = 17780
s.add(TSP <= 17790) #Maximum range of trip setpoint (count) = 17790
s.add(TSP >= 999) #Minimum range of trip setpoint (count) = 999
s.add(TRIP_LOGIC == False) #initially trip signal is off.

# Definition of FBD program logic
__tbool = False
ENO__1__RESET_FALLING = False
OUT__1__RESET_FALLING = False
ENO__2__RESET_FALLING = False
OUT__2__RESET_FALLING = 0
ENO__3__RESET_FALLING = False
OUT__3__RESET_FALLING = False
ENO__4__RESET_FALLING = False
OUT__4__RESET_FALLING = 0

ENO__1__RESET_FALLING, OUT__1__RESET_FALLING = LE_REAL(True, PV_OUT, TSP,
ENO__1__RESET_FALLING, OUT__1__RESET_FALLING)
__tbool = If(True, And(OUT__1__RESET_FALLING,Not(TRIP_LOGIC)), __tbool)
ENO__2__RESET_FALLING, OUT__2__RESET_FALLING = WORD_TO_REAL(True, HYS,
ENO__2__RESET_FALLING, OUT__2__RESET_FALLING)
ENO__3__RESET_FALLING, OUT__3__RESET_FALLING = MOVE_BOOL(__tbool, True,
ENO__3__RESET_FALLING, OUT__3__RESET_FALLING)
TRIP_LOGIC1 = If(__tbool, OUT__3__RESET_FALLING, TRIP_LOGIC)
ENO__4__RESET_FALLING, OUT__4__RESET_FALLING = ADD2_REAL(__tbool, TSP,
OUT__2__RESET_FALLING, ENO__4__RESET_FALLING, OUT__4__RESET_FALLING)
TSP1 = If(__tbool, OUT__4__RESET_FALLING, TSP)

# -- Define test requirement
s.add(TRIP_LOGIC1 == True)

# -- check satisfiability & return a solution for the model
print '* satisfiability of model (sat/un-sat): %s' %s.check()
print '* found model: ', s.model()

```

length : 1,914 lines : 47 Ln : 47 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS



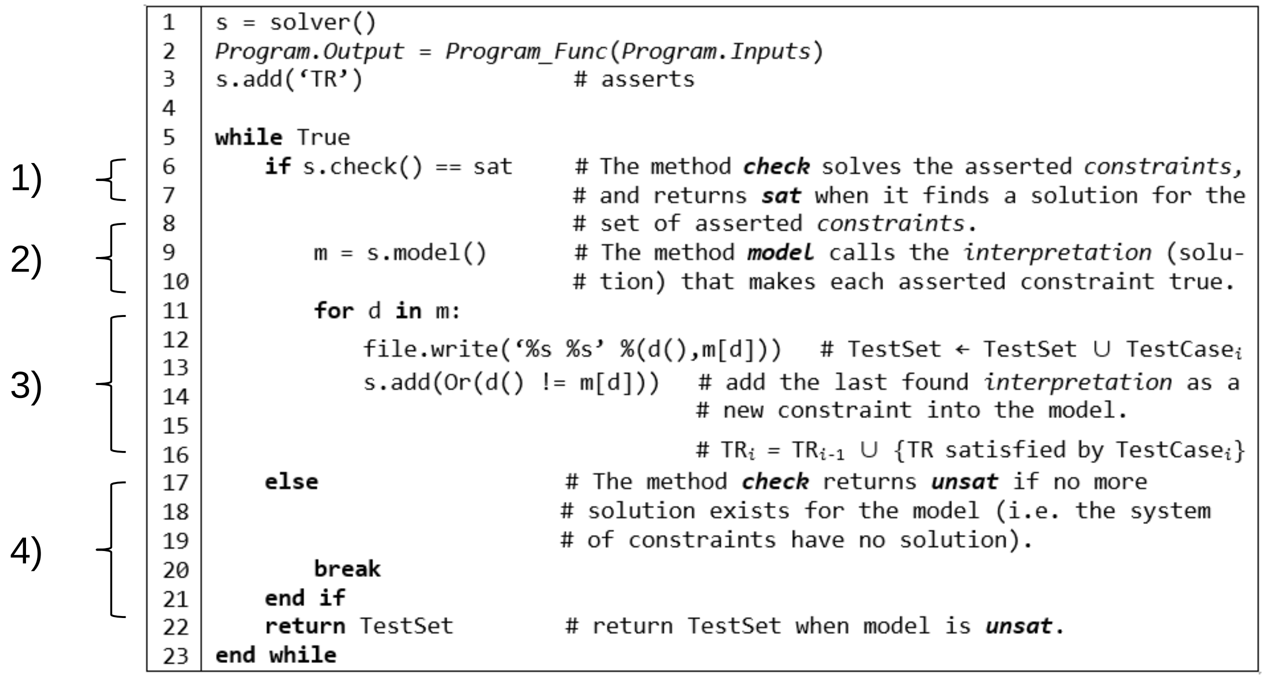
```
D:₩Cppexample₩WindowsFormsApplication1_test_20181119₩WindowsFormsApp
lication1_test₩bin₩Debug₩resultfiles>python main.py
* satisfiability of model (sat/un-sat): sat
* found model:  [PV_OUT = 17780, TSP_R = 17780, TRIP_LOGIC = False]

D:₩Cppexample₩WindowsFormsApplication1_test_20181119₩WindowsFormsApp
lication1_test₩bin₩Debug₩resultfiles>
```

Screenshot of single test case generation algorithm execution

Z3 input file for single test case generation of FBD program

Appendix X.

- Exhaustive Test Case Generation for NPP Safety Software
  - To derive all interpretations (solutions) to FBD program, at each iteration, the algorithm:
    - 1) Derives the model from satisfiability check
    - 2) Saves the model as a single test case
    - 3) Adds a new constraint that negates the last found interpretation to the test requirement at each iteration.
    - 4) If the formula is *unsatisfiable*, return the derived model as exhaustive test cases.
      - *Unsatisfiable* means there exist no interpretation (solution) that evaluates a given formula to true under given constraints.



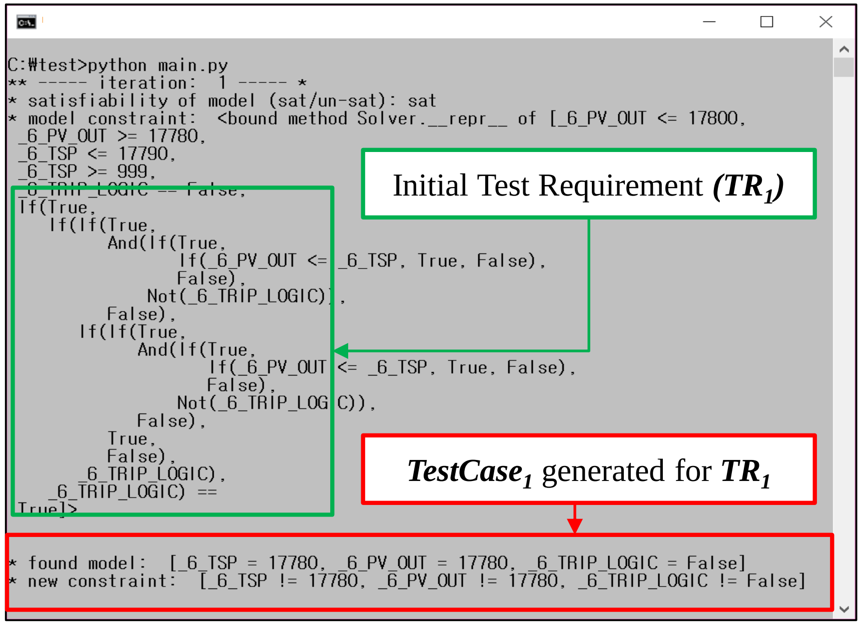
```
s = solver()
Program.Output = Program_Func(Program.Inputs)
s.add('TR')                    # asserts

while True
    if s.check() == sat        # The method check solves the asserted constraints,
                               # and returns sat when it finds a solution for the
                               # set of asserted constraints.
        m = s.model()          # The method model calls the interpretation (solu-
                               # tion) that makes each asserted constraint true.
        for d in m:
            file.write('%s %s' %(d(),m[d]))   # TestSet ← TestSet ∪ TestCase_i
            s.add(Or(d() != m[d]))   # add the last found interpretation as a
                                     # new constraint into the model.

                                     # TR_i = TR_{i-1} ∪ {TR satisfied by TestCase_i}
    else                       # The method check returns unsat if no more
                               # solution exists for the model (i.e. the system
                               # of constraints have no solution).
        break
    end if
    return TestSet             # return TestSet when model is unsat.
end while
```

Exhaustive test case generation algorithm for FBD programs

Appendix XI.

- Example of Exhaustive Test Case Generation for Simple FBD Program

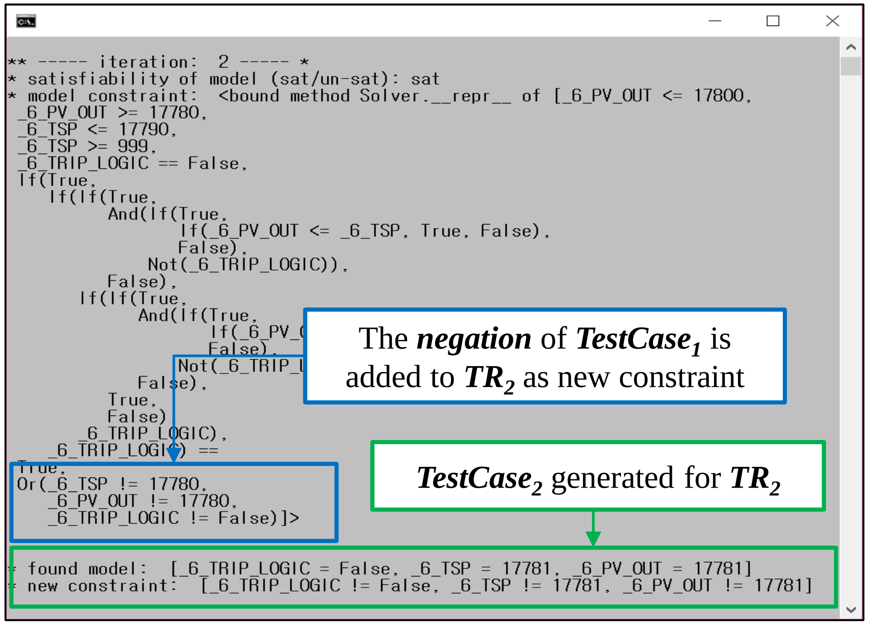
1st Iteration result


Initial Test Requirement *(TR₁)*
TestCase₁ generated for *TR₁*

```
C:\test>python main.py
** ----- iteration:  1 ----- *
* satisfiability of model (sat/un-sat): sat
* model constraint:  <bound method Solver.__repr__ of [_6_PV_OUT <= 17800,
 _6_PV_OUT >= 17780,
 _6_TSP <= 17790,
 _6_TSP >= 999,
 _6_TRIP_LOGIC == False,
 If(True,
    If(If(True,
          And(If(True,
                 If(_6_PV_OUT <= _6_TSP, True, False),
                 False),
              Not(_6_TRIP_LOGIC)),
          False),
       If(If(True,
             And(If(True,
                    If(_6_PV_OUT <= _6_TSP, True, False),
                    False),
                 Not(_6_TRIP_LOGIC)),
             False),
          True,
          False),
       _6_TRIP_LOGIC),
    _6_TRIP_LOGIC) ==
 True]>

* found model:  [_6_TSP = 17780, _6_PV_OUT = 17780, _6_TRIP_LOGIC = False]
* new constraint:  [_6_TSP != 17780, _6_PV_OUT != 17780, _6_TRIP_LOGIC != False]
```

***TestCase*$_1$** = [PV_OUT = 17780, TSP = 17780, TRIP_LOGIC = False]

2nd Iteration result


The ***negation*** of ***TestCase₁*** is added to ***TR₂*** as new constraint
TestCase₂ generated for *TR₂*

```
** ----- iteration:  2 ----- *
* satisfiability of model (sat/un-sat): sat
* model constraint:  <bound method Solver.__repr__ of [_6_PV_OUT <= 17800,
 _6_PV_OUT >= 17780,
 _6_TSP <= 17790,
 _6_TSP >= 999,
 _6_TRIP_LOGIC == False,
 If(True,
    If(If(True,
          And(If(True,
                 If(_6_PV_OUT <= _6_TSP, True, False),
                 False),
              Not(_6_TRIP_LOGIC)),
          False),
       If(If(True,
             And(If(True,
                    If(_6_PV_
                    False),
                 Not(_6_TRIP_L
             False),
          True,
          False)
       _6_TRIP_LOGIC),
    _6_TRIP_LOGIC) ==
 True,
 Or(_6_TSP != 17780,
    _6_PV_OUT != 17780,
    _6_TRIP_LOGIC != False)]>

* found model:  [_6_TRIP_LOGIC = False, _6_TSP = 17781, _6_PV_OUT = 17781]
* new constraint:  [_6_TRIP_LOGIC != False, _6_TSP != 17781, _6_PV_OUT != 17781]
```

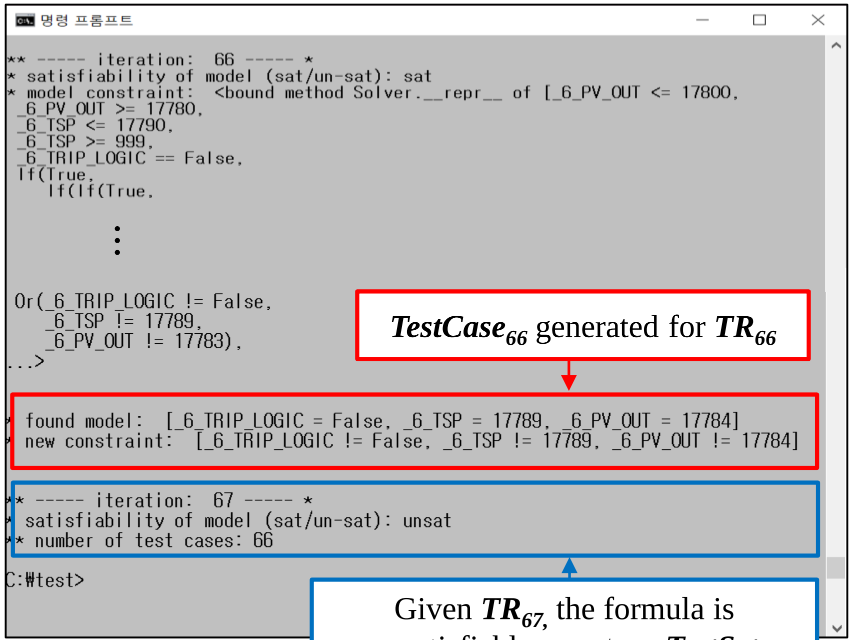
***TestCase*$_2$** = [PV_OUT = 17781, TSP = 17781, TRIP_LOGIC = False]

Execution result of exhaustive test case generation algorithm for an example PLC program

Appendix XI.

- Example of Exhaustive Test Case Generation for Simple FBD Program

66th and 67th Iteration result

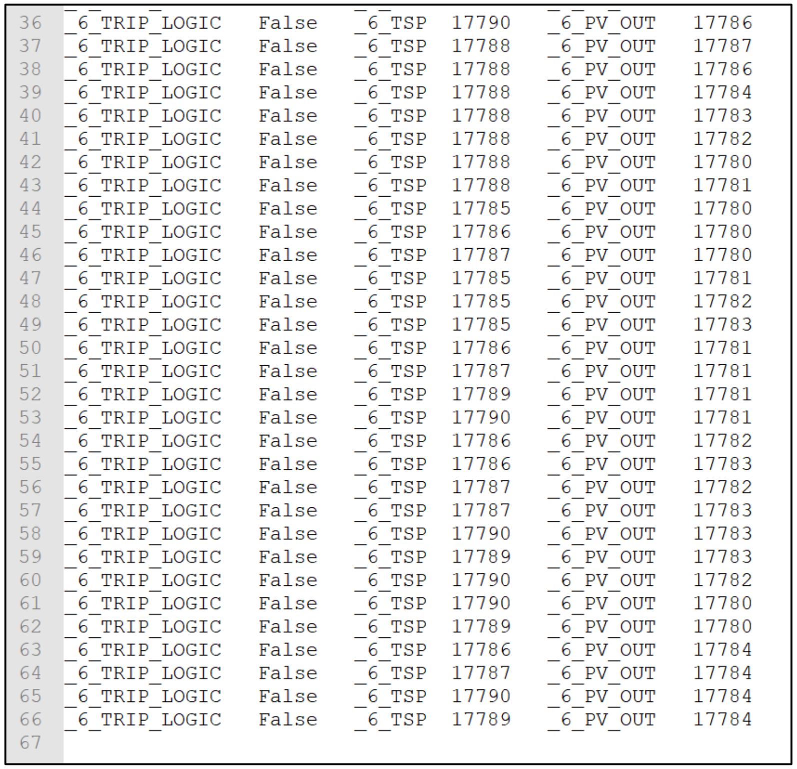


```
** ----- iteration:  66 ----- *
* satisfiability of model (sat/un-sat): sat
* model constraint:  <bound method Solver.__repr__ of [_6_PV_OUT <= 17800,
 _6_PV_OUT >= 17780,
 _6_TSP <= 17790,
 _6_TSP >= 999,
 _6_TRIP_LOGIC == False,
 If(True,
    If(If(True,

          ⋮

 Or(_6_TRIP_LOGIC != False,
     _6_TSP != 17789,
     _6_PV_OUT != 17783),
...>

* found model:  [_6_TRIP_LOGIC = False, _6_TSP = 17789, _6_PV_OUT = 17784]
* new constraint:  [_6_TRIP_LOGIC != False, _6_TSP != 17789, _6_PV_OUT != 17784]

** ----- iteration:  67 ----- *
* satisfiability of model (sat/un-sat): unsat
** number of test cases: 66

C:₩test>
```

TestCase $_{66}$ generated for ***TR*** $_{66}$

Given ***TR*** $_{67}$, the formula is unsatisfiable → return ***TestSet***

TestCase $_{66}$ = [PV_OUT = 17784, TSP = 17789, TRIP_LOGIC = False]



```
36  _6_TRIP_LOGIC   False   _6_TSP  17790   _6_PV_OUT   17786
37  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17787
38  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17786
39  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17784
40  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17783
41  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17782
42  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17780
43  _6_TRIP_LOGIC   False   _6_TSP  17788   _6_PV_OUT   17781
44  _6_TRIP_LOGIC   False   _6_TSP  17785   _6_PV_OUT   17780
45  _6_TRIP_LOGIC   False   _6_TSP  17786   _6_PV_OUT   17780
46  _6_TRIP_LOGIC   False   _6_TSP  17787   _6_PV_OUT   17780
47  _6_TRIP_LOGIC   False   _6_TSP  17785   _6_PV_OUT   17781
48  _6_TRIP_LOGIC   False   _6_TSP  17785   _6_PV_OUT   17782
49  _6_TRIP_LOGIC   False   _6_TSP  17785   _6_PV_OUT   17783
50  _6_TRIP_LOGIC   False   _6_TSP  17786   _6_PV_OUT   17781
51  _6_TRIP_LOGIC   False   _6_TSP  17787   _6_PV_OUT   17781
52  _6_TRIP_LOGIC   False   _6_TSP  17789   _6_PV_OUT   17781
53  _6_TRIP_LOGIC   False   _6_TSP  17790   _6_PV_OUT   17781
54  _6_TRIP_LOGIC   False   _6_TSP  17786   _6_PV_OUT   17782
55  _6_TRIP_LOGIC   False   _6_TSP  17786   _6_PV_OUT   17783
56  _6_TRIP_LOGIC   False   _6_TSP  17787   _6_PV_OUT   17782
57  _6_TRIP_LOGIC   False   _6_TSP  17787   _6_PV_OUT   17783
58  _6_TRIP_LOGIC   False   _6_TSP  17790   _6_PV_OUT   17783
59  _6_TRIP_LOGIC   False   _6_TSP  17789   _6_PV_OUT   17783
60  _6_TRIP_LOGIC   False   _6_TSP  17790   _6_PV_OUT   17782
61  _6_TRIP_LOGIC   False   _6_TSP  17790   _6_PV_OUT   17780
62  _6_TRIP_LOGIC   False   _6_TSP  17789   _6_PV_OUT   17780
63  _6_TRIP_LOGIC   False   _6_TSP  17786   _6_PV_OUT   17784
64  _6_TRIP_LOGIC   False   _6_TSP  17787   _6_PV_OUT   17784
65  _6_TRIP_LOGIC   False   _6_TSP  17790   _6_PV_OUT   17784
66  _6_TRIP_LOGIC   False   _6_TSP  17789   _6_PV_OUT   17784
67
```

Generated Exhaustive Test Cases (***TestSet***)

Execution result of exhaustive test case generation algorithm for an example PLC program