Verification of Function Block Diagram through Verilog Translation

Seungjae Jeon, Eunkyoung Jee, Sungdeok Cha[1],
Junbeom Yoo[2], and Geeyoung Park[3]

[1] Div. of Computer Science, Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
{sjjeon,ekjee,cha}@dependable.kaist.ac.kr
[2] Samsung Electronics Co., Ltd.
Suwon, Republic of Korea
junbeom.yoo@samsung.com
[3] Korea Atomic Energy Research Institute,
Daejeon, Republic of Korea
gypark@kaeri.re.kr

Abstract. The formal verification of FBD program is required at nuclear power plant as traditional relay-based analog systems are being replaced with digital PLC based software. This paper proposes a way to formally verify the FBD program. For this purpose, Verilog model is automatically translated from the FBD program, then Cadence SMV performs model checking. We demonstrated the effectiveness of the suggested approach by conducting a case study of the nuclear reactor protection system, which is currently being developed in Korea.

1 Introduction

Software safety became a critical issue in nuclear power plant area because traditional analog systems are being replaced by Programmable Logic Controller (PLC) based software[5]. As formal methods are gaining acceptance in research community as a promising approach to provide a high degree of safety assurance, several formal specification and verification methods have been developed and applied to nuclear power plant systems.

KNICS[3] consortium is developing a suite of instrumentation and control software for next generation Korean nuclear power plants, which is classified as being safety-critical by government regulation authority. Currently being developed advanced power reactor's (APR-1400) protection system (RPS) is thoroughly verified using formal verification technique such as model checking[6].

PLC is a special type of industrial computer largely used in control systems. It provides powerful functionality to deal with periodic time and polling mechanism. International Electrotechnical Commission (IEC) defined five application software programming languages for PLCs. Among them, Function Block Diagram (FBD) is one of the most widely used languages. A major part of KNICS APR-1400 RPS Software Design Specification (SDS)[4] is specified in FBD.

Rigorous safety demonstration is required on FBD program since it is automatically compiled to machine code and executed on industrial computers. Correctness of FBD program can be guaranteed by using formal verification technique as well as traditional testing and simulation methods.

This paper proposes a way to formally verify FBD program. We define the syntax of FBD formally based on the IEC standard, then translate the program into Verilog[10] model. Translated Verilog model is verified using Cadence SMV[14] model checker. APR-1400 RPS is used as a case study to show effectiveness of the proposed approach.

A tool, *FBD2V*, is implemented to support proposed approach. It generates Verilog model from FBD program. It is also used as a front-end for model checking and counterexample analysis. These features enable nuclear engineers to verify FBD program with minimum expertise on formal method.

The remainder of the paper is organized as follows: section 2 explains FBD, Verilog, and Cadence SMV briefly. Section 3 describes the translation rules from FBD to Verilog. Section 4 presents FBD2V and a case study of a real system. Section 5 presents related works, and section 6 concludes this paper.

2 Background

2.1 PLC programming in FBD

PLC is an industrial computer applied to wide range of control systems. The main characteristic of PLC program is *scan cycle*[8]. In each iteration of this permanent loops, the program reads inputs, computes new internal states, and updates outputs. This cyclic behavior makes PLCs suitable for interacting with a continuous environment.

FBD is one of the standard PLC programming languages identified in IEC61131-3[7]. FBD is widely used because of its graphical notations and usefulness in applications with a high degree of data flow among control components. FBD defines system behavior in terms of flow of signals among function blocks. A collection of function blocks is wired together in a manner of a circuit diagram.

Fig.1 shows ten function block groups and a representative example of each group. Arithmetic, comparison, bitwise boolean, type conversion, selection, and numerical blocks do not have internal states. They always produce a primary value as a result when executed with a particular set of input values. In contrast, timer, edge detection, bistable and counter blocks store values in internal and output variables[9].

Fig.2 gives an example of FBD to calculate TRIP_T and TSP. The outputs are produced by the sequential combination of the block operations. Details will be explained with formal definitions in next section.

2.2 Verilog

Verilog[10] is one of the most popular Hardware Description Languages (HDL) used by integrated circuit (IC) designers. Below we summarize the Verilog features[2] pertinent to our discussion.

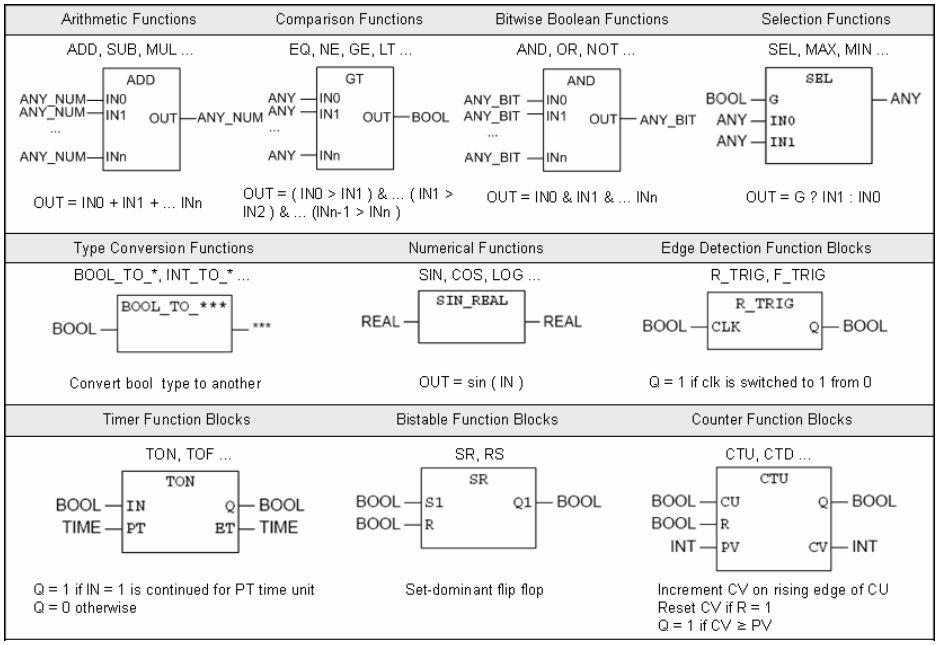


Fig. 1. Function block groups and representative examples

Verilog has several types of variables. A **wire** variable represents a physical wire in a circuit and is used to connect gates or modules. A wire does not store its value, and must be driven by the **assign** statement or by connected output of a gate or a module. On the other hand, a **reg** variable is a data object holding its value. Reg variables are assigned only in **always** and **initial** block.

A module is a principal design entry in Verilog. Module declaration specifies the name and list of I/O ports. The first part of a module defines I/O and data type of each port. Keywords **input** and **output** declare the input and output ports of a module. Data type is specified for each variable, e.g., as the size of a bit vector. Module declarations are templates from that one creates actual instantiations. Modules are instantiated inside other modules and each instantiation creates a unique object from the template. The exception is top-level module (i.e., main) which is its own instantiation.

2.3 Model Checking and Cadence SMV

We use model checking technique to formally verify FBD programs. Model checking is a technique to prove whether a formal system satisfies certain properties or not. Cadence SMV is a model checker based on symbolic model checking technique[12]. Cadence SMV can verify a model programmed in Synchronous Verilog (SV)[11], a slight variation of the Verilog language with cycle-based behavior. Cadence SMV converts Synchronous Verilog into SMV input language[13], and

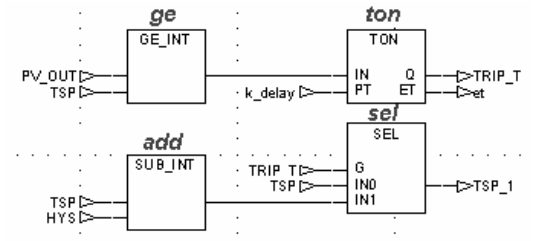


Fig. 2. A small example of FBD

then performs model checking. *True* is returned if Verilog model meets given property. Otherwise, a *counterexample* is produced to show the existence of errors in the model.

3 Verilog Translation from FBD

An FBD program is translated to a Verilog model in order to perform model checking by Cadence SMV. This section mainly describes how to translate an FBD program into a Verilog model. First subsection formally defines function blocks and function block diagrams. Those definitions are based on [1] and slightly modified. Next subsection restricts the scope of target FBD program. Then we show translation steps with a small example.

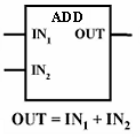
3.1 Formal Definition of FBD

Defintion 1 (FB Type) *Function block type is defined as a tuple* $< Type\_name,\ IP,\ OP,\ BD >$*, where*

- $Type\_name$*: a name of function block type*
- IP*: a set of input ports,* $\{IP_1, \ldots, IP_M\}$
- OP*: a set of output ports,* $\{OP_1, \ldots, OP_N\}$
- BD*: behavior description, as functions for each* OP*,* $BD_{OP_n} : (IP_1, \ldots, IP_M) \rightarrow OP_n\ , 1 \leq n \leq N$ ⌟

Input port (IP) and output port (OP) are the official term used in the standard [7]. Fig.3 describes an example of ADD block. Other function blocks can be defined in the similar way.

As FBD is a network of function blocks, each block is considered as an *instance* of function block type. Instance names of blocks are specified in Fig.2; *ge*, *ton*, *add*, and *sel*. We write *sel.G* to indicate the port named G in block *sel* for convenience. Behavior description of function block instance is written similarly; $add.BD_{OUT}(add.IN_1, add.IN_2) = add.IN_1 + add.IN_2$.



< **ADD** ,
{ **IN1, IN2** } ,
{ **OUT** } ,
{ **BDOUT** (**IN1** , **IN2**) = **IN1** + **IN2** } >

Fig. 3. Example: Formal definition of ADD block

Defintion 2 (FBD) *FBD is defined as a tuple* $< FBs,\ V,\ T >$*, where*

- FBs*: a set of function block instances*
- V*: a set of input and output variables of FBD,* $V = V_I \bigcup V_O$
  - V_I*: a set of input variables into FBD*
  - V_O*: a set of output variables from FBD*
- T*: a set of transitions between* FBs *and* V
  $V_I \times FB.IP \bigcup FB.OP \times FB.IP \bigcup FB.OP \times V_O$ ⌟

Let V_O be a set of output variables computed at each iteration of scan cycles. V_I is a set of input variables and each $v_i \in V_I$ has its own value; their values are constants, set by external, or output variables having same name. Transition T represents wires connecting variables and function blocks. Fig.4 shows the example FBD formally defined.

$< FBs,\ V,\ T >$

$FBs = \{ge, ton, add, sel\}$
V_I = {PV_OUT, TSP, k_delay, HYS, TRIP_T}
V_O = {TRIP_T, et, TSP_1}
T = { (PV_OUT , $ge.IN_1$) , (TSP_1 , $add.IN_1$) ,
(TSP , $ge.IN_2$) , (HYS , $add.IN_2$) ,
($ge.OUT$, $ton.IN$) , (TRIP_T , $sel.G$) ,
(k_delay , $ton.PT$) , (TSP , $sel.IN_0$) ,
($ton.Q$, TRIP_T) , ($add.OUT$, $sel.IN_1$) ,
($ton.ET$, et) , ($sel.OUT$, TSP_1) }

Fig. 4. Formal definition of Fig.2

Defintion 3 (Evaluation function) *Each port and variables are evaluated as* f*: (port or variable)* $\rightarrow$ *FBD_data_type*

- *For input variable* $p_0 \in V_I$*,* $f(p_0) = p_0$
- *For output variable* $p_{recv} \in V_O$*, or input port* $p_{recv} \in fb.IP, fb \in FBs$*, let* $(p' \times p_{recv}) \in T$*,* $f(p_{recv}) = f(p')$
- *For output port* $p_{emit} \in fb.OP, fb \in FBs$*, let* $fb.IP = \{p_1, ..., p_M\}$*,* $f(p_{emit}) = fb.BD_p\{p_1, ..., p_M\}$ ⌟

Output variables in FBD are evaluated by inputs and function blocks connected. For example, TSP_1 at Fig.2 is evaluated as below:
$f(\text{TSP\_1}) = f(sel.OUT)$
$= sel.BD_{OUT}(f(sel.G), f(sel.IN_0), f(sel.IN_1))$
$= f(sel.G)\ ?\ f(sel.IN_1)\ :\ f(sel.IN_0)$
$= \text{TRIP\_T}\ ?\ add.BD_{OUT}(f(add.IN_1), f(add.IN_2))\ :\ \text{TSP}$
$= \text{TRIP\_T}\ ?\ (f(add.IN_1) + f(add.IN_2))\ :\ \text{TSP}$
$= \text{TRIP\_T}\ ?\ (\ \text{TSP} + \text{HYS}\ )\ :\ \text{TSP}$

3.2 Assumptions on FBD

FBD should satisfy following assumptions in order to be translated into Verilog. These assumptions correspond to FBD semantics stated in IEC 61131-3 standard.

FBD is well wired

- Every port and variable are connected.
  $\{x|(x \times y) \in T\} = \{p_{emit}|p_{emit} \in V_I \text{ or } p_{emit} \in fb.OP, fb \in FBs\}$
  $\{y|(x \times y) \in T\} = \{p_{recv}|p \in V_O \text{ or } p_{recv} \in fb.IP, fb \in FBs\}$
- Every port and variable has only one source.
  $\forall(x \times y) \in T\ \forall x' \neq x,\ (x' \times y) \notin T$

FBD is type safe

- $\forall(x \times y) \in T$, x and y should have same data type; e.g., bool, int, or word. FBD data type is defined in the standard.

FBD should not overwrite output variables

- Every output variable has unique name so that its value can be assigned only once per cycle. Some FBD development tools allow overwriting output variables. In this case, output variables should be renamed to temporary names to be distinguished from each other.

Execution order is predefined

- Output variables are evaluated in given order. Let an ordered set $V_O = \{v_{o1}, ..., v_{oN}\}$, computation starts from v_{o1} and ends at v_{oN} within a cycle.

3.3 Translation Steps

If FBD program satisfies all the assumptions, it is ready to be translated into Verilog model. Each steps will be explained with an example FBD program shown in Fig.2.

```
//Rule 1. module declaration:
module main (clk, [input_variables], [output_variables]);

// Rule 2. for each variable v ∈ V:
input | reg | wire | output [size(v) : 0] v;

initial begin
// Rule 3. for each reg variable v_reg:
v_reg <= [initial_value_of_v_reg];
end

// Rule 4. for each wire and output variable v_o ∈ V_O:
assign v_o = f(v_o);

always @ (posedge clk) begin
// Rule 5. for each reg variable v_reg:
v_reg <= [stored_value];
end

always [if ([condition])] assert [label]: [assertion]; // property
endmodule
```

Fig. 5. Verilog generation template

Variable type detection

Each variable in FBD is mapped to one of Verilog variable types; input, reg, wire and output. A input variable $v_i \in V_I$ is **input** type if there is no output variable having same name with v_i, i.e., its value is transmitted from external input. v_i is **reg** type if its value needs to be stored internally. Reg variables hold their value and will be used at next cycle operation. On the other hand, values that need to be stored just for this cycle are declared as **wire** variables. They represent physical wires connecting function blocks and variables. A output variable $v_o \in V_O$ is **output** type if it is designated as an external output of the module. In Fig.2, variables are classified as following:

- V_{input} = {PV_OUT, TSP, k_delay, HYS}
- V_{wire} = {TRIP_T, et, TSP_1}

Variable size decision

Non-boolean values are represented as bit vectors and their size should be decided. We use notation $size(v)$ for number of bit size required to represent v. Let $size(v) = 0$ if v is boolean variable. Size of input and reg variables should be given by the user so that a model checker can cover proper range of the input variables. Size of wire and output variables are computed from the connected inputs, reg variables and function blocks. They should be large enough to represent maximum values in the program.

Let $size$(PV_OUT) = $size$(TSP) = 7, $size$(k_delay) = 4, $size$(HYS) = 2 given by user, then

- $size$(TRIP_T) = 0
- $size$(et) = $size$(k_delay) = 4
- $size$(TSP_1) = max($size$(TSP), $size$(HYS)) + 2 = 9

Output variable assignment

A Verilog expression for assigning a variable p has a same semantic with $f(p)$ at definition 3. Function blocks that do not store internal states are mapped to Verilog operators. Examples of blocks having internal states are timers, flip-flops, and counters. These function blocks are translated into Verilog modules.

- f(TSP_1) = TRIP_T ? (TSP + HYS) : TSP
- f(TRIP_T) = $ton.BD_Q$(PV_OUT >= TSP, k_delay),
  behavior of TON is translated into Verilog module as shown in Fig.6.

```
module main (clk, PV_OUT, TSP, k_delay, HYS);
  input clk;
  input [7:0] PV_OUT, TSP;
  input [4:0] k_delay;
  input [2:0] HYS;
  wire TRIP_T;
  wire [4:0] et;
  wire [9:0] TSP_1;
  TON ton (clk, (PV_OUT >= TSP), k_delay, TRIP_T, et);
  assign TSP_1 = TRIP_T ? ( TSP + HYS ) : TSP;
endmodule

module TON (clk, IN, PT, Q, ET);
  input clk, IN;
  input [4:0] PT;
  output Q;
  output [4:0] ET;
  reg [4:0] t;
  initial t = 0;
  assign ET = t;
  assign Q = IN && (ET >= PT);
  always @ (posedge clk)
    t <= IN ? ((t < PT) ? t+1 : PT) : 0;
endmodule
```

Fig. 6. Verilog model translated from Fig.2

Generation of Verilog model

Based on translation rules in [2], Verilog model is generated according to the template described in Fig.5. In Rule 1, module name, input and output ports are specified in the first line. Variables are declared with their type, bit size, and name in Rule 2. Rule 3 initiates the **reg** variables. The main evaluation logic, expressed by a collection of function blocks and variables in FBD, is translated by Rule 4. Stored values are assigned to reg variables in Rule 5. **@** (**posedge** clk) means positive edge of clk signal, i.e., the beginnings of each cycle. As updated value of a reg variable becomes visible at next time unit, new value is read at next cycle[14]. Finally, properties are embedded by the user.

– Verilog model Fig.6 is generated from Fig.2 by applying Rule 1 - 5.

4 Case Study

This section demonstrates a case study of the proposed FBD verification technique. Target system is Bistable Processor (BP) at APR-1400 RPS[4]. An FBD program which is a part of BP, is translated into Verilog model through the rules introduced in previous section. Then, the tool FBD2V is briefly introduced. Model checking result of the program is analyzed using FBD2V, and then we show how an error is discovered.

Fig.7 shows FIX_RISING program, one of the modules in BP system. It sets the variaable TRIP_LOGIC_out when the variable PV_OUT exceeds certain range for given time units. The output TRIP_LOGIC_out takes part in the emergency shutdown logic of nuclear reactor. The FBD is well wired, type safe, does not overwrite output variables, and has top-down (traditional) execution order. It satisfies all the assumptions.

To translate FIX_RISING program into Verilog model, we detect variable type first. As PV_OUT, HYS, and MAXCNT are appeared only in input variables V_I, they are **input** type. TRIP_CNT, TRIP_LOGIC, and TSP are **reg** type variables whose values are stored and used at next cycle. TRIP_CNT_out, TSP_1, TSP_out, TRIP_LOGIC_1, and TRIP_LOGIC_out appear in both input and output variable set. Their values are assigned in wires and become inputs for evaluating other variables, but they are not stored for next cycle, i.e., **wire** type. Other steps are as same as explained in previous section.

Fig.8 shows Verilog model generated from FIX_RISING program through Rule 1 - 5 in Fig.5. System specification defines that HYS, MAXCNT, and TSP have non-zero initial values; they are hard coded in line 16 - 23. Two properties are embedded in line 37 - 40. Property A1 means that "Trip should be set if TRIP_CNT_out becomes larger than or equal to MAXCNT." A2 means that "Trip should be reset if PV_OUT is less than or equal to TSP_out."

FBD2V automates suggested FBD verification framework. It takes LDA file as input then converts it into a Verilog model. LDA file is a FBD storing format of a tool pSET[15] used by KNICS consortium. User adjusts bit size and initial values of the variables during the translation, as shown in Fig.9. After the

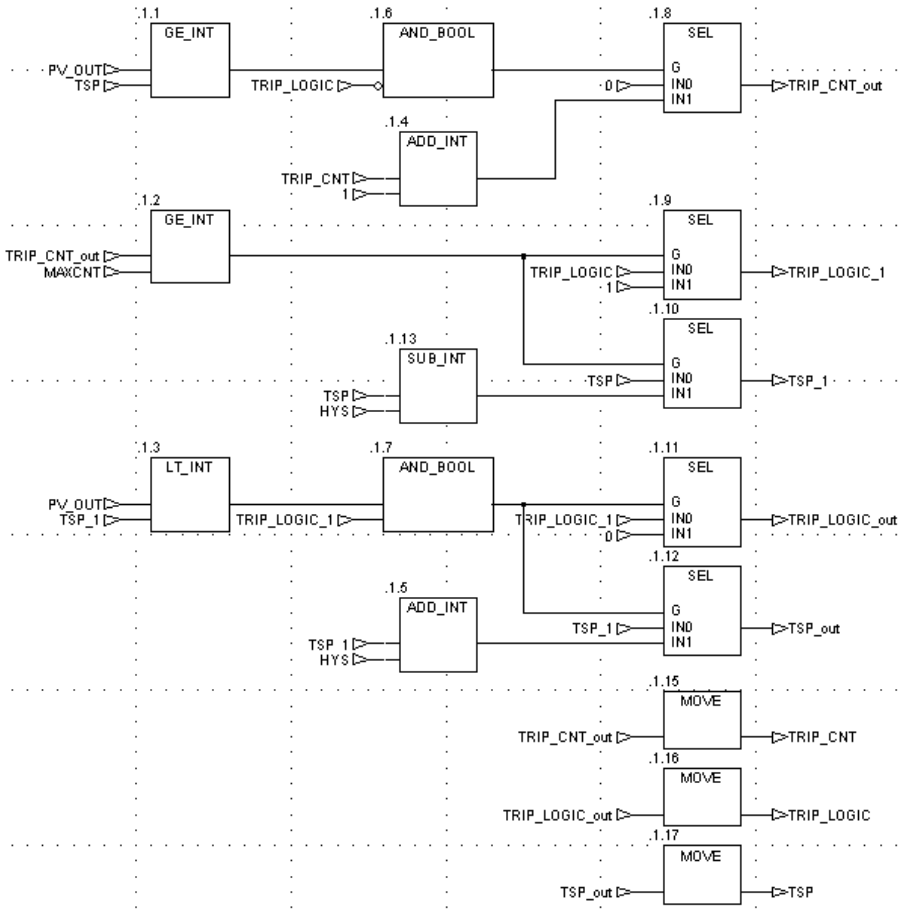


Fig. 7. FIX_RISING program

properties are embedded, FBD2V executes Candence SMV and model checking is performed. To enhance readability of the counterexample, it is displayed in timing graph form, which is familiar to hardware engineers. Variables are highlighted in different colors and shapes for visualization.

Fig.10 shows model checking result of FIX_RISING program displayed in FBD2V. The left side is Cadence SMV result and the right side is a timing graph representation of the counterexample. The program *failed* to satisfy the property A2, "Trip should be reset if PV_OUT is less than or equal to TSP_out." To aid counterexample analysis, FBD2V enables users to declare monitoring variables; constants, variables in counterexample, and arithmetic operators can be used to declare a monitoring variable. PV_OUT $\leq$ TSP_out is displayed at the bottom of the figure to check the condition of the property. Although this condition is satisfied at the 6th cycle, TRIP_LOGIC_out holds the same value

```
1  module main (clk, HYS, MAXCNT, PV_OUT);
2
3  input clk;
4  input [2:0] HYS;
5  input [4:0] MAXCNT;
6  input [7:0] PV_OUT;
7  reg [4:0] TRIP_CNT;
8  reg TRIP_LOGIC;
9  reg [7:0] TSP;
10 wire [5:0] TRIP_CNT_out;
11 wire TRIP_LOGIC_1;
12 wire [7:0] TSP_1;
13 wire TRIP_LOGIC_out;
14 wire [8:0] TSP_out;
15
16 assign HYS = 1;
17 assign MAXCNT = 5;
18
19 initial begin
20 TRIP_CNT <= 0;
21 TRIP_LOGIC <= 0;
22 TSP <= 20;
23 end
24
25 assign TRIP_CNT_out =
        ((PV_OUT >= TSP) && ! TRIP_LOGIC) ? (TRIP_CNT + 1) : 0;
26 assign TRIP_LOGIC_1 =
        (TRIP_CNT_out >= MAXCNT) ? 1 : TRIP_LOGIC;
27 assign TSP_1 =
        (TRIP_CNT_out >= MAXCNT) ? (TSP - HYS) : TSP;
28 assign TRIP_LOGIC_out =
        ((PV_OUT < TSP_1) && TRIP_LOGIC_1) ? 0 : TRIP_LOGIC_1;
29 assign TSP_out =
        (((PV_OUT < TSP_1) && TRIP_LOGIC_1) ? (TSP_1 + HYS) : TSP_1;
30
31 always @ (posedge clk) begin
32 TRIP_CNT <= TRIP_CNT_out;
33 TRIP_LOGIC <= TRIP_LOGIC_out;
34 TSP <= TSP_out;
35 end
36
37 always begin
38 if ( TRIP_CNT_out >= MAXCNT ) assert A1: TRIP_LOGIC_out == 1;
39 if ( TRIP_LOGIC && PV_OUT =< TSP_out )
        assert A2: TRIP_LOGIC_out == 0;
40 end
41 endmodule
```

Fig. 8. FIX_RISING program translated into Verilog

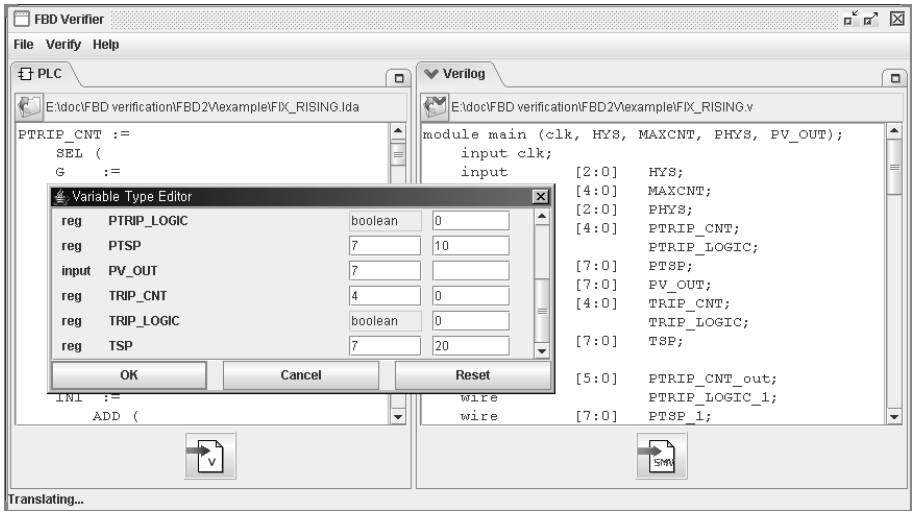


Fig. 9. FBD2V screenshot

with TRIP_LOGIC_1. As a result, the logic assigning TRIP_LOGIC_out has an error. User can conclude that the LT_INT block is misused instead of LE_INT block.

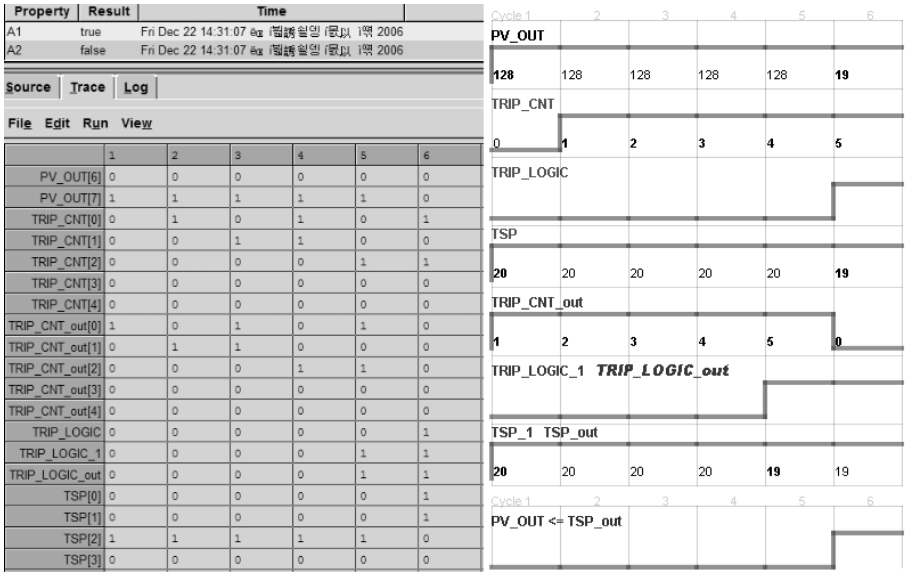
BP is composed of 6 modules, including FIX_RISING program, and 18 drip decision logic implementations. Each trip decision logics uses two or more modules with specific parameters, just like a function call. Every logic and module was formally verified with proposed method, and errors were found.

There was a state explosion problem with the program having large number of inputs or storing variables for long term of cycles. We adopted a manual abstraction technique to make the verification feasible. Automated abstraction and slicing techniques for Verilog model will be needed for futurework.

5 Related Work

Verilog translation from FBD and verification technique was previously proposed in [2]. It focused on mechanical generation of FBD from NuSCR formal specification and equivalence checking using VIS verifier[16] on various versions of FBD programs. It originally devised Verilog translation rules in order to use VIS. It also stated the possibility for model checking on the translated Verilog program. Main difference of our research is that we developed a tool to automate the FBD model checking framework. Verilog translation rules are modified to generate Verilog code. Visualized counterexample analysis is another contribution of our tool FBD2V.

There are other many Verilog HDL model checkers. CBMC[18] checks Verilog for consistency with ANSI-C program. VCEGAR[19] performs model checking of



	1	2	3	4	5	6
PV_OUT[6]	0	0	0	0	0	0
PV_OUT[7]	1	1	1	1	1	0
TRIP_CNT[0]	0	1	0	1	0	1
TRIP_CNT[1]	0	0	1	1	0	0
TRIP_CNT[2]	0	0	0	0	1	1
TRIP_CNT[3]	0	0	0	0	0	0
TRIP_CNT[4]	0	0	0	0	0	0
TRIP_CNT_out[0]	1	0	1	0	1	0
TRIP_CNT_out[1]	0	1	1	0	0	0
TRIP_CNT_out[2]	0	0	0	1	1	0
TRIP_CNT_out[3]	0	0	0	0	0	0
TRIP_CNT_out[4]	0	0	0	0	0	0
TRIP_LOGIC	0	0	0	0	0	1
TRIP_LOGIC_1	0	0	0	0	1	1
TRIP_LOGIC_out	0	0	0	0	1	1
TSP[0]	0	0	0	0	0	1
TSP[1]	0	0	0	0	0	1
TSP[2]	1	1	1	1	1	0
TSP[3]	0	0	0	0	0	0

Fig. 10. A counterexample from FIX_RISING program

Verilog using CounterExample Guided Abstraction Refinement[20] framework. Using these model checkers instead of Cadence SMV might be meaningful.

Counter-example visualization is one of the active research areas. smv2vcd[17] converts SMV counterexample into industrial standard format, Variable Change Dump (VCD). VCD file can be viewed and analyzed by a wide variety of tools.

6 Conclusion

This paper proposed a method for formal verification of FBD. We suggested a way to automatically translate Verilog model from FBD program. The generated Verilog model is verified with Cadence SMV model checker. Verilog model generation and counterexample analysis are done with tool support.

Contributions of suggested method are followings: First, FBD program is thoroughly verified by model checking using automated tool FBD2V. Second, FBD2V aids analysis of counterexamples computed by Cadence SMV. FBD2V represents a counterexample in timing graph form which is familiar to hardware engineers. User can declare monitoring variables and slice variables to debug the FBD program.

The proposed method was applied to the verification of KNICS APR-1400 RPS. Several errors were found and they were noticed to nuclear engineers to be fixed in the next revision.

Acknowledgment This work was supported by the Korea Science and Engineering Foundation(KOSEF) through the Advanced Information Technology Research

Center(AITrc). This work was also supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement)

References

1. Junbeom Yoo, Hojung Bang, Sungdeok Cha. FBD Program Synthesis for PLC Controllers. Science of Computer Programming, 2005, submitted.
2. Junbeom Yoo. Synthesis of Function Block Diagrams from NuSCR Formal Specification. Doctoral Thesis, 2005.
3. KNICS(Korea Nuclear Instrumentation and Control System Research and Development Center), http://www.knics.re.kr/english/eindex.html
4. Korea Atomic Energy Research Institute. SDS for reactor protection system. KNICS-RPS-SDS231 Rev.02, 2006.
5. U. NRC. Digital Instrumentation and Control Systems in Nuclear Power Plants: safety and reliability issues. National Academy Press, 1997.
6. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, 1999.
7. IEC. International Standard for Programmable Controllers: Programming Languages. Part 3, 1993.
8. A. Mader. A Classification of PLC Models and Applications. In Proc. WODES 2000: 5th Workshop on Discrete Event Systems, August 21-23, Gent, Belgium, 2000.
9. R. Lewis. Programming industrial control systems using IEC 1131-3 Revised Edition(IEE Control Engineering Series). The Institute of Electrical Engineers, 1998.
10. IEEE Standard Hardware Description Language Based on the Verilog hardware Description Language (IEEE Std 1364-2001). IEEE, 2003.
11. Ching-Tsun Chou. Synchronous Verilog: A Proposal. Fujitsu Laboratories of America, 1997.
12. K.L.McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.
13. vl2smv manual, http://www.cis.ksu.edu/santos/smv-doc/vl2smvman.txt
14. Cadence SMV tutorial, http://www.cis.ksu.edu/santos/smv-doc/tutorial/tutorial.html
15. pSET (POSCON Software Engineering Tool), http://rnd.poscon.co.kr
16. VIS, http://vlsi.colorado.edu/∼vis/
17. smv2vcd, http://www.cs.cmu.edu/∼modelcheck/smv2vcd.html
18. Bounded Model Checking for ANSI-C, http://www.cs.cmu.edu/∼modelcheck/cbmc/
19. H. Jain, N. Sharygina, D. Kroening, E. Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In 42nd Design Automation Conference, 2005.
20. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5), 2003.