

Verilog 변환을 이용한 FBD의 정형검증

Verification of Function Block Diagram through
Verilog Translation



Verification of Function Block Diagram through Verilog Translation

Advisor : Professor Cha, Sungdeok

by

Jeon, Seungjae

Department of Electrical Engineering and Computer Science

Division of Computer Science

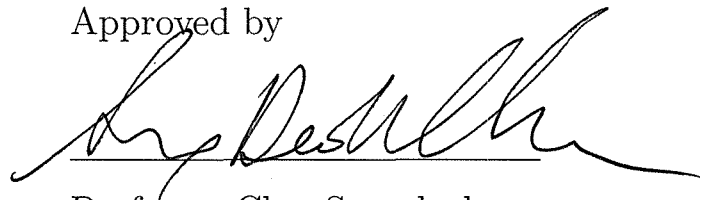
Korea Advanced Institute of Science and Technology

A thesis submitted to the faculty of the Korea Advanced
Institute of Science and Technology in partial fulfillment of the
requirements for the degree of Master of Engineering in the
Department of Electrical Engineering and Computer Science,
Division of Computer Science

Daejeon, Korea

2006. 12. 22.

Approved by



Professor Cha, Sungdeok

Advisor

Verilog 변환을 이용한 FBD의 정형검증

전 승재

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

KAIST

2006년 12월 12일

심사위원장 차 성 덕



심사위원 이 윤 준



심사위원 황 규 영



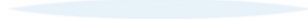
MCS 전 승재. Jeon, Seungjae. Verification of Function Block Diagram through Verilog
20053532 Translation . Verilog 변환을 이용한 FBD의 정형검증. Department of Electrical
Engineering and Computer Science, Division of Computer Science . 2007. 26p.
Advisor Prof. Cha, Sungdeok. Text in English.

Abstract

The formal verification of FBD program is required in nuclear engineering domain 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.



KAIST



Contents

| | |
|---|------------|
| Abstract | i |
| Contents | iii |
| List of Figures | v |
| 1 Introduction | 1 |
| 2 Background | 3 |
| 2.1 PLC programming in FBD | 3 |
| 2.2 Verilog | 3 |
| 2.3 Model Checking and Cadence SMV | 4 |
| 3 Verilog Translation from FBD | 6 |
| 3.1 Formal Definition of FBD | 6 |
| 3.2 Assumptions on FBD | 9 |
| 3.2.1 FBD is well wired | 9 |
| 3.2.2 FBD is type safe | 9 |
| 3.2.3 FBD should not overwrite output variables | 9 |
| 3.2.4 Execution order is predefined | 9 |
| 3.3 Translation Steps | 10 |
| 3.3.1 Variable type detection | 10 |
| 3.3.2 Variable size decision | 11 |
| 3.3.3 Output assignment to Verilog expression | 11 |
| 3.3.4 Verilog generation | 12 |
| 4 Case Study | 15 |
| 4.1 FIX_RISING example | 15 |
| 4.2 FBD2V | 17 |
| 5 Related Work | 22 |

| | |
|----------------------------|-----------|
| 6 Conclusion | 23 |
| Summary (in Korean) | 24 |
| References | 25 |



List of Figures

| | | |
|-----|---|----|
| 2.1 | Function block groups and examples | 4 |
| 2.2 | FBD Example | 5 |
| 3.1 | FBD Verification Framework | 6 |
| 3.2 | Example: Formal definition of ADD block | 7 |
| 3.3 | Example: Fig.2.2 formally defined | 8 |
| 3.4 | Example: FBD not satisfying assumptions | 10 |
| 3.5 | Verilog generation template | 13 |
| 3.6 | Example: Fig.2.2 translated into Verilog model | 14 |
| 4.1 | Original FIX_RISING program | 15 |
| 4.2 | FIX_RISING program without overwriting output variables | 16 |
| 4.3 | FIX_RISING program translated into Verilog | 18 |
| 4.4 | FBD2V screenshot | 19 |
| 4.5 | Counterexample from FIX_RISING | 20 |
| 4.6 | FBD2V feature: counterexample sliced by user | 21 |
| 5.1 | FBD generation and equivalence framework by [2] | 22 |

1. Introduction

Software safety became a critical issue in nuclear engineering 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 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.

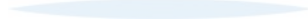
Rigorous safety demonstration is required on FBD program since it is automatically compiled to machine code and executed on industrial computers. Correctness of FBD program can be guaranteed by using formal verification technique as well as traditional testing and simulation methods.

This paper proposes a way to formally verify FBD program. We define the 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 enables 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 introduces related

KAIST



2. Background

2.1 PLC programming in FBD

Programmable Logic Controller (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.

Function Block Diagram (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[1]. 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.2.1 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.2.2 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.

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

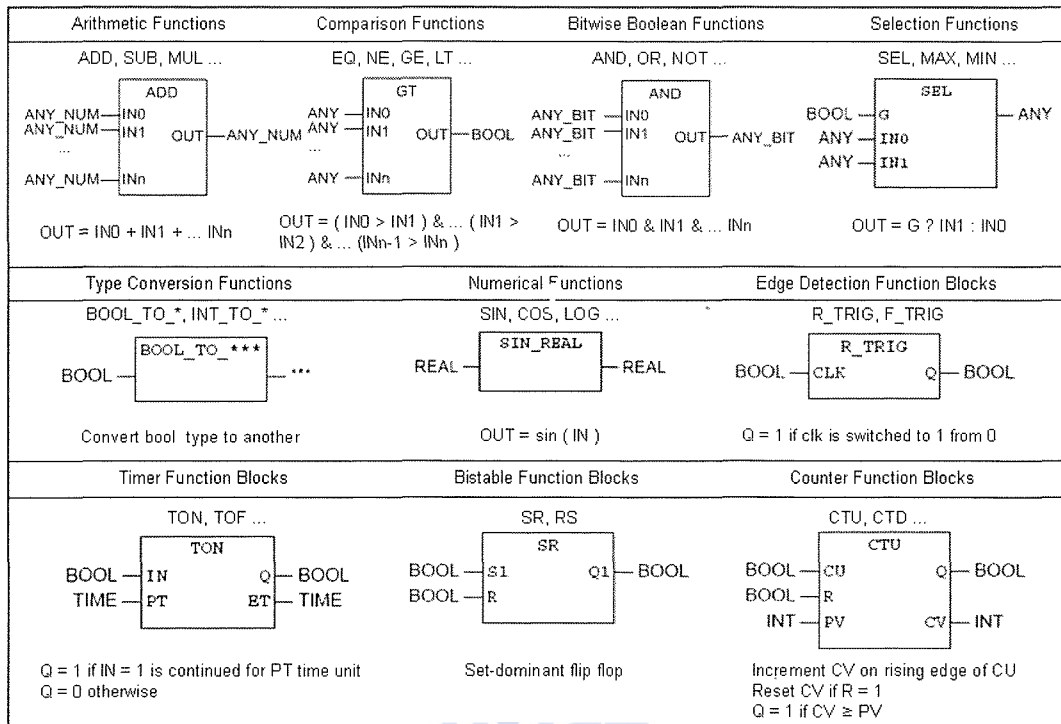


Figure 2.1: Function block groups and examples

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 generally represented as the size of a bit vector. Module declarations are templates from which 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 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

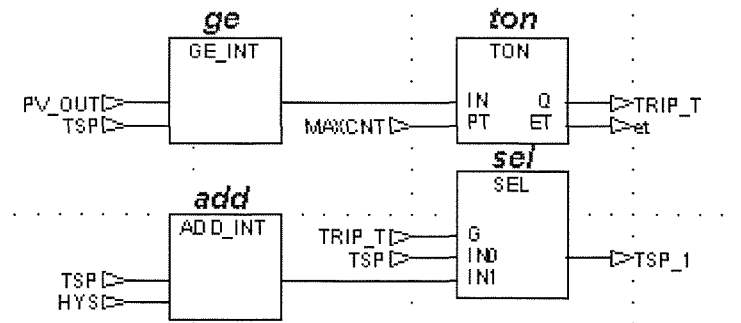


Figure 2.2: FBD Example

cycle-based behavior. It converts Synchronous Verilog into SMV input language[13], and then performs model checking. *True* is returned if Verilog model meets given properties. Otherwise, a *counterexample* is produced to show the existence of errors in the model.



3. Verilog Translation from FBD

Fig.3.1 shows an overview of FBD verification framework. Verilog model is translated from target FBD program. Properties, specifications of the system, are embedded in Verilog model as assertions[14]. Cadence SMV performs model checking on the Verilog model, then counterexample is analyzed.

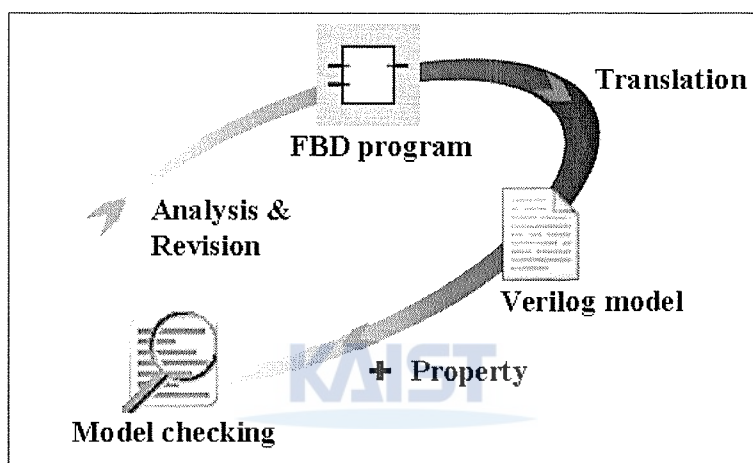


Figure 3.1: FBD Verification Framework

This section mainly describes how to translate FBD program into 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 $\langle Type, IP, OP, BD \rangle$, where*

- *Type: a name of function block type*
- *IP: a set of input ports, $\{IP_1, \dots, IP_M\}$*

- *OP*: a set of output ports, $\{OP_1, \dots, OP_N\}$
- *BD*: behavior description, as functions for each *OP*,
 $BD_{OP_n} : (IP_1, \dots, IP_M) \rightarrow OP_n, 1 \leq n \leq N$ \lrcorner

Input port (IP) and output port (OP) are the official term used in the standard [7]. Fig.3.2 describes an example of ADD block. Other function blocks can be defined in the similar way.

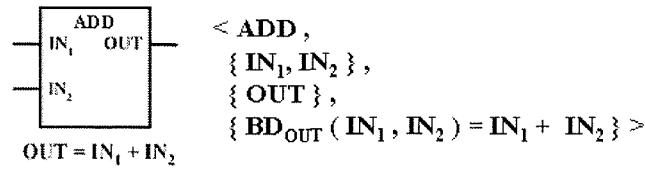


Figure 3.2: Example: Formal definition of ADD block

As FBD is a network of function blocks, we can consider each block as an *instance* of function block type. Instance names of blocks are specified in Fig.2.2; *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; $add.BD_{OUT}(add.IN_1, add.IN_2) = add.IN_1 + add.IN_2$.

Defintion 2 (FBD) *Function block diagram is defined as a tuple $\langle FBs, V, T \rangle$, where*

- *FBs*: a set of function block instances
- *V*: a set of input and output variables of FBD,
 $V = V_I \cup V_O$
 - 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
 - $V_I \times FB.IP$
 - $FB.OP \times FB.IP$
 - $FB.OP \times V_O$ \lrcorner

$$\begin{aligned}
& \langle FBs, V, T \rangle \\
& FBs = \{ge, ton, add, sel\} \\
& V_I = \{PV_OUT, TSP, MAXCNT, HYS, TRIP_T\} \\
& V_O = \{TRIP_T, et, TSP_1\} \\
& T = \{ PV_OUT \times ge.IN_1, TSP_1 \times add.IN_1, \\
& \quad TSP \times ge.IN_2, HYS \times add.IN_2, \\
& \quad ge.OUT \times ton.IN, TRIP_T \times sel.G, \\
& \quad MAXCNT \times ton.PT, TSP \times sel.IN_0, \\
& \quad ton.Q \times TRIP_T, add.OUT \times sel.IN_1, \\
& \quad ton.ET \times et, sel.OUT \times TSP_1 \}
\end{aligned}$$

Figure 3.3: Example: Fig.2.2 formally defined

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 set by external, output variables having same name, or constants. Transition T represents wires connecting variables and function blocks. Fig.3.3 shows the FBD example formally defined.



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

- For input variable $p \in V_I$,
 $f(p) = p$
- For output variable $p \in V_O$
or input port of a block $p \in fb.IP, fb \in FBs$,
let $(p' \times p) \in T$,
 $f(p) = f(p')$
- For output port of a block $p \in fb.OP, fb \in FBs$,
let $fb.IP = \{p_1, \dots, p_M\}$,
 $f(p) = fb.BD_p\{p_1, \dots, p_M\}$ \lrcorner

Output variables in FBD are evaluated by inputs and function blocks connected. For example, TSP_1 at Fig.2.2 is evaluated as below:

$$f(TSP_1) = f(sel.OUT)$$

$$\begin{aligned}
&= sel.BD_{OUT}(f(sel.G), f(sel.IN_0), f(sel.IN_1)) \\
&= f(sel.G) ? f(sel.IN_1) : f(sel.IN_0) \\
&= TRIP_T ? add.BD_{OUT}(f(add.IN_1), f(add.IN_2)) : TSP \\
&= TRIP_T ? (f(add.IN_1) + f(add.IN_2)) : TSP \\
&= TRIP_T ? (TSP + HYS) : TSP
\end{aligned}$$

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.

3.2.1 FBD is well wired

- Every port and variable is connected.
$$\{x|(x \times y) \in T\} = \{p|p \in V_I \text{ or } p \in fb.OP, fb \in FBs\}$$

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

3.2.2 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.

3.2.3 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.

3.2.4 Execution order is predefined

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

Fig.3.4 shows examples of FBD not satisfying the assumptions. As IN1 in a. has two sources, bool1 and bool2, it is vague that which input variable should be selected. IN2 is not connected, therefore this AND block cannot be computed. b. presents an example of unmatched type; integer value cannot be negated. c. describes that an output variable *c* is overwritten by two blocks. If top-down execution order is predefined in this case, two *c*s are renamed to *c_1* and *c_2*, respectively.

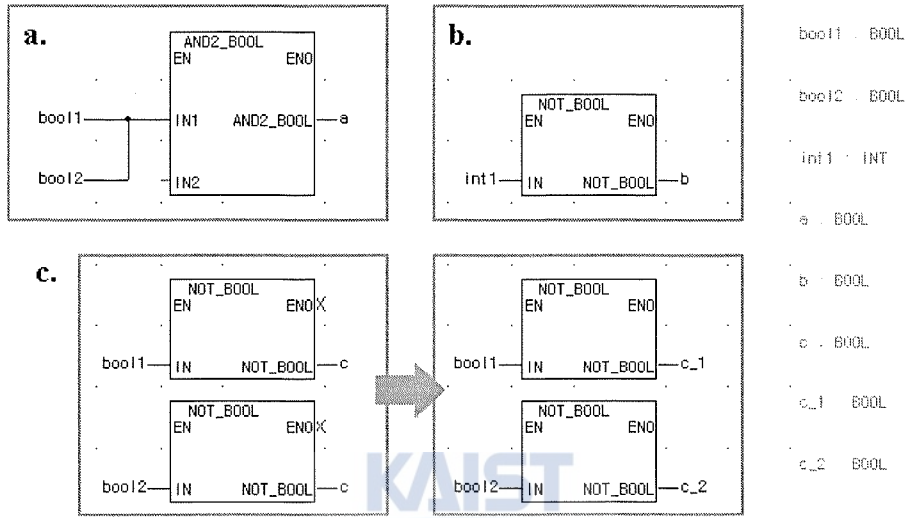


Figure 3.4: Example: FBD not satisfying assumptions

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.2.2.

3.3.1 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. 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 assign to **wire** variable. They represents 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.

- $V_{input} = \{PV_OUT, TSP, MAXCNT, HYS\}$
- $V_{wire} = \{TRIP_T, et, TSP_1\}$

3.3.2 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 input range of the program. Size of wire and output variables are computed from the connected input, reg variables and function blocks. They should be large enough to represent maximum values in the program.

- Let $size(PV_OUT) = size(TSP) = 7$, $size(MAXCNT) = 4$, $size(HYS) = 2$ given by user
- $size(TRIP_T) = 0$
- $size(et) = size(MAXCNT) = 4$
- $size(TSP_1) = \max(size(TSP), size(HYS)) + 1 = 8$

3.3.3 Output assignment to Verilog expression

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

- $f(TSP_1) = TRIP_T ? (TSP + HYS) : TSP$

- $f(\text{TRIP_T}) = \text{ton.BD}_Q(\text{PV_OUT} \geq \text{TSP}, \text{MAXCNT})$,
behavior of TON is translated into Verilog module.

3.3.4 Verilog generation

Based on translation rules in [2], Verilog model is generated as Fig.3.5. 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 the beginnings of each cycle. As updated value of a reg variable becomes visible at next time unit[14], new value is read at next cycle. Finally, properties are embedded by the user.

- Verilog model is generated as Fig.3.6 from the FBD program through Rule 1 - 5.

The logo for KAIST (Korea Advanced Institute of Science and Technology) is centered on the page. It consists of the letters "KAIST" in a bold, blue, sans-serif font. Below the text is a light blue, horizontal oval shape that tapers at both ends, resembling a stylized shadow or a base.

```

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

// Rule 2. for each variable  $v \in V$ :
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  $v_o \in V_O$ :
assign  $v_o = f(v_o)$ ;

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

always begin
// properties
if ([condition]) assert [label]: [assertion];
end

endmodule

```

Figure 3.5: Verilog generation template

```

module main (clk, PV_OUT, TSP, MAXCNT, HYS);
  input clk;
  input [7:0] PV_OUT;
  input [7:0] TSP;
  input [4:0] MAXCNT;
  input [1:0] HYS;
  wire TRIP_T;
  wire [4:0] et;
  wire [8:0] TSP_1;
  // instantiation of module TON
  TON ton (clk, (PV_OUT >= TSP), MAXCNT, TRIP_T, et);
  assign TSP_1 = TRIP_T ? ( TSP + HYS ) : TSP;
endmodule

module TON (clk, IN, PT, Q, ET);
  input clk;
  input 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

```

Figure 3.6: Example: Fig.2.2 translated into Verilog model

4. Case Study

4.1 FIX_RISING example

This section demonstrates an example of a real system translated into Verilog model. Target system is Bistable Processor (BP) at APR-1400 RPS[4]. BP consists of several decision logics of trip, emergency shutdown of nuclear reactor.

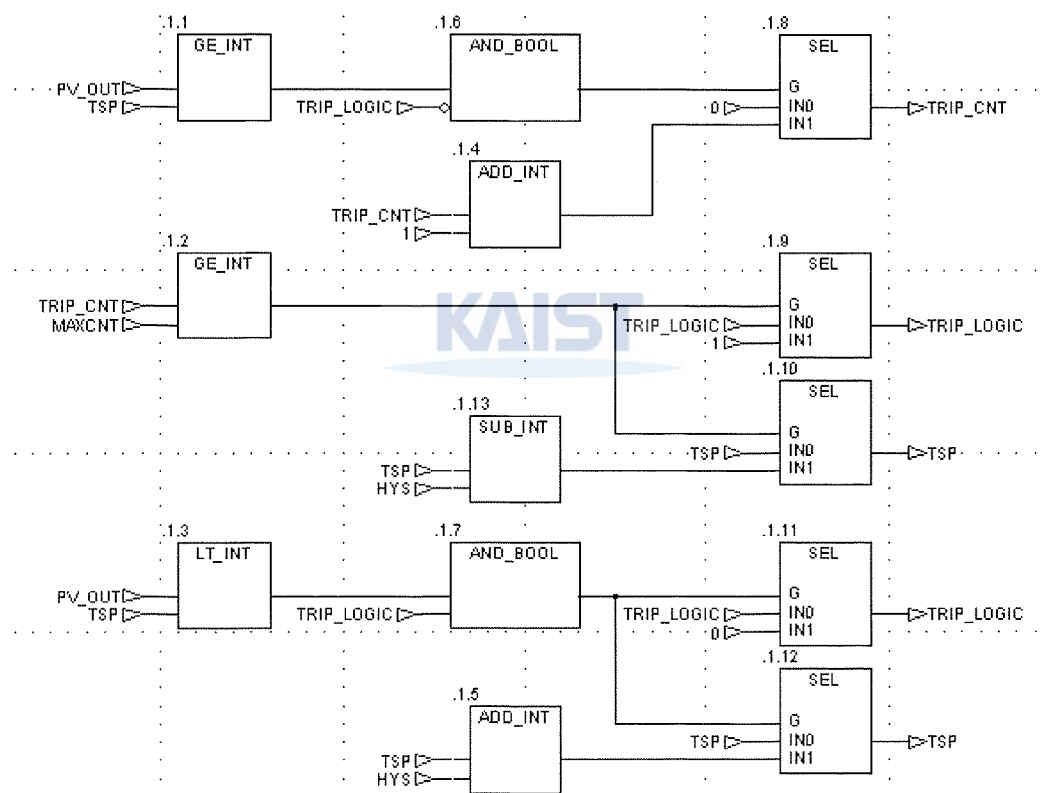


Figure 4.1: Original FIX_RISING program

Fig.4.1 shows FIX_RISING program, one of the modules in BP. The output TRIP_LOGIC_out takes part in the trip decision logic.

The FBD is well wired, type safe, and has top-down (traditional) execution order. But

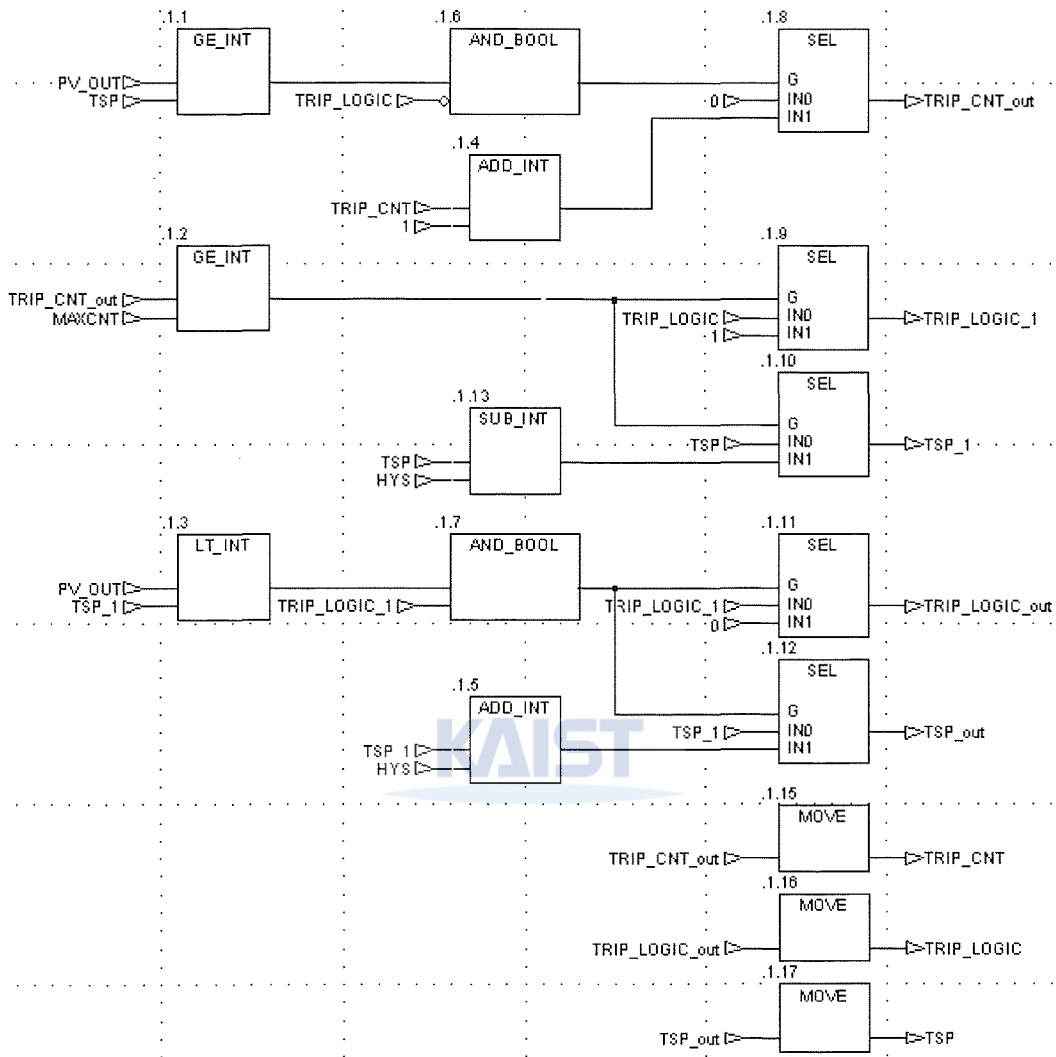


Figure 4.2: FIX_RISING program without overwriting output variables

it overwrites output variables; TRIP_CNT, TRIP_LOGIC, and TSP. To make FBD satisfy the assumptions, it is processed as Fig.4.2. Duplicated output variables are renamed to have postfix, "_1", "_out", etc., in order to distinguished from each other. Values storage logics for next cycle are explicitly specified using MOVE blocks.

To translate Fig.4.2 into Verilog, we detect variable type first. As {PV_OUT, HYS, MAXCNT} are appeared only in input variables V_I , they are **input** type. {TRIP_CNT, TRIP_LOGIC, TSP} are **reg** type variables which are stored and used at next cycle.

{TRIP_CNT_out, TSP_1, TSP_out, TRIP_LOGIC_1, TRIP_LOGIC_out} are appeared both in input and output variables (V_I and V_O). 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.

We decide bit size of the variables next. Let $size(PV_OUT) = 7$, $size(HYS) = 2$, $size(MAXCNT) = size(TRIP_CNT) = 4$ are given by the user. $size(TRIP_CNT_out) = \max(0, size(TRIP_CNT) + 1) = 5$ because maximum value produced by SEL block is the largest one between IN0 and IN1, and ADD block merges the range of inputs. Similarly, $size(TSP_1) = 7$, $size(TSP_out) = 8$. SMV will give warnings if the variables exceed their range.

To see an example of definition 3, TRIP_LOGIC_1 in Fig.3.3 is evaluated as below.

$$\begin{aligned}
f(TRIP_LOGIC_1) &= f(SEL2.OUT) \\
&= SEL2.BD_{OUT}(f(SEL2.G), f(SEL2.IN0), f(SEL2.IN1)) \\
&= f(SEL2.G) ? f(SEL2.IN1) : f(SEL2.IN0) \\
&= GE2.BD_{OUT}(f(GE2.IN1), f(GE2.IN2)) ? 1 : TRIP_LOGIC \\
&= (TRIP_CNT_out \geq MAXCNT) ? 1 : TRIP_LOGIC
\end{aligned}$$

Fig.4.3 shows Verilog model generated from FIX_RISING program through Rule 1 - 5.

System specification defines that HYS, MAXCNT, and TSP have non-zero initial values; they are hard coded in the Verilog model. Two properties are embedded in the example. A1 means "Trip should be set if TRIP_CNT_out becomes larger than or equal to MAXCNT." A2 means "Trip should be unset if PV_OUT is less than or equal to TSP_out."

4.2 FBD2V

This section demonstrates the usefulness of the proposed formal verification technique. A tool, *FBD2V*, is implemented to support the verification framework. We briefly introduce the tool, explain how to analyze the model checking result of FIX_RISING program, and then discover an error.

FBD2V automates the FBD verification framework described in Fig.3.1. It takes LDA file, FBD storing format of a tool[15] used by KNICS consortium, as input then converts the FBD into Verilog model. User adjusts bit size and initial values of the variables during the translation, as shown in fig.4.4. After properties are embedded, FBD2V executes Candence SMV and model checking result is computed. To enhance readability of counterexample, it is displayed in timing graph form, which is familiar to hardware engineers. Variables are highlighted in different color and shape for visualization.

Fig.4.5 shows model checking result of FIX_RISING program displayed in FBD2V. Right

```

module main (clk, HYS, MAXCNT, PV_OUT);

input clk;
input [2:0] HYS;
input [4:0] MAXCNT;
input [7:0] PV_OUT;
reg [4:0] TRIP_CNT;
reg TRIP_LOGIC;
reg [7:0] TSP;
wire [5:0] TRIP_CNT_out;
wire TRIP_LOGIC_1;
wire [7:0] TSP_1;
wire TRIP_LOGIC_out;
wire [8:0] TSP_out;
//constants
assign HYS = 1;
assign MAXCNT = 5;

initial begin
TRIP_CNT <= 0;
TRIP_LOGIC <= 0;
TSP <= 20;
end

assign TRIP_CNT_out = ((PV_OUT >= TSP) && ! TRIP_LOGIC) ? (TRIP_CNT + 1) : 0;
assign TRIP_LOGIC_1 = (TRIP_CNT_out >= MAXCNT) ? 1 : TRIP_LOGIC;
assign TSP_1 = (TRIP_CNT_out >= MAXCNT) ? (TSP - HYS) : TSP;
assign TRIP_LOGIC_out = ((PV_OUT < TSP_1) && TRIP_LOGIC_1) ? 0 : TRIP_LOGIC_1;
assign TSP_out = (((PV_OUT < TSP_1) && TRIP_LOGIC_1) ? (TSP_1 + HYS) : TSP_1;

always @ (posedge clk) begin
TRIP_CNT <= TRIP_CNT_out;
TRIP_LOGIC <= TRIP_LOGIC_out;
TSP <= TSP_out;
end

always begin
if ( TRIP_CNT_out >= MAXCNT ) assert A1: TRIP_LOGIC_out == 1;
if ( TRIP_LOGIC == 1 && PV_OUT =< TSP_out ) assert A2: TRIP_LOGIC_out == 0;
end

endmodule

```

Figure 4.3: FIX_RISING program translated into Verilog

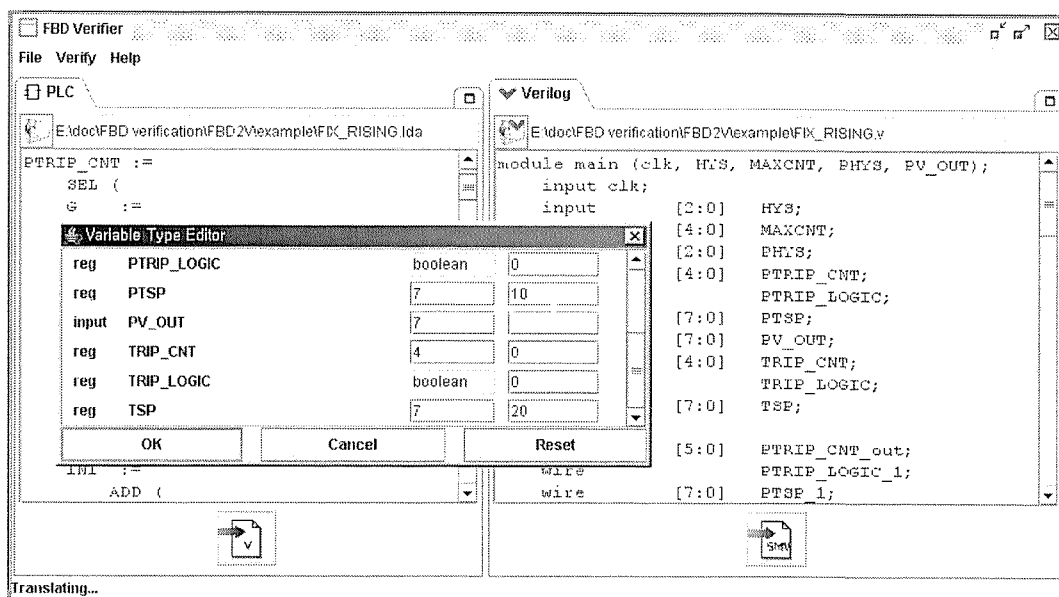


Figure 4.4: FBD2V screenshot

side is original counterexample shown by Cadence SMV and left side is timing graph representation. The program *failed* to satisfy the property A2, "Trip should be unset if PV_OUT is less than or equal to TSP_out." To aid counterexample analysis, the tool enables users to declare monitoring variables; constants, variables in counterexample, and arithmetic operators can be used. 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 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 has six modules including FIX_RISING introduced above. 18 trip decision logics are implemented in BP, where two or more modules are interleaved to compute each logic. Every logic and module was formally verified with proposed framework, and errors were found.

Fig.4.6 shows a counterexample from a trip logic containing FIX_RISING module. It is composed of approximately 40 blocks and 20 variables. Because of its size, counterexample is more complicated than fig.4.5. FBD2V supports variable slicing for user to hide variables. After slicing, as shown in right side of the figure, its error cause is as same as appeared in fig.4.5.

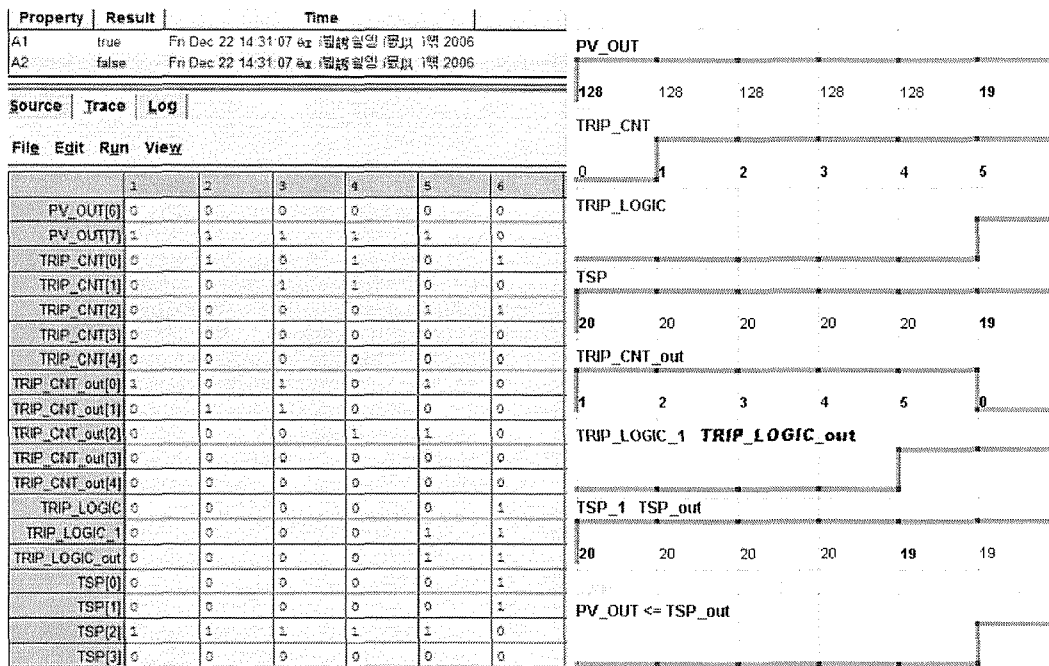


Figure 4.5: Counterexample from FIX_RISING

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

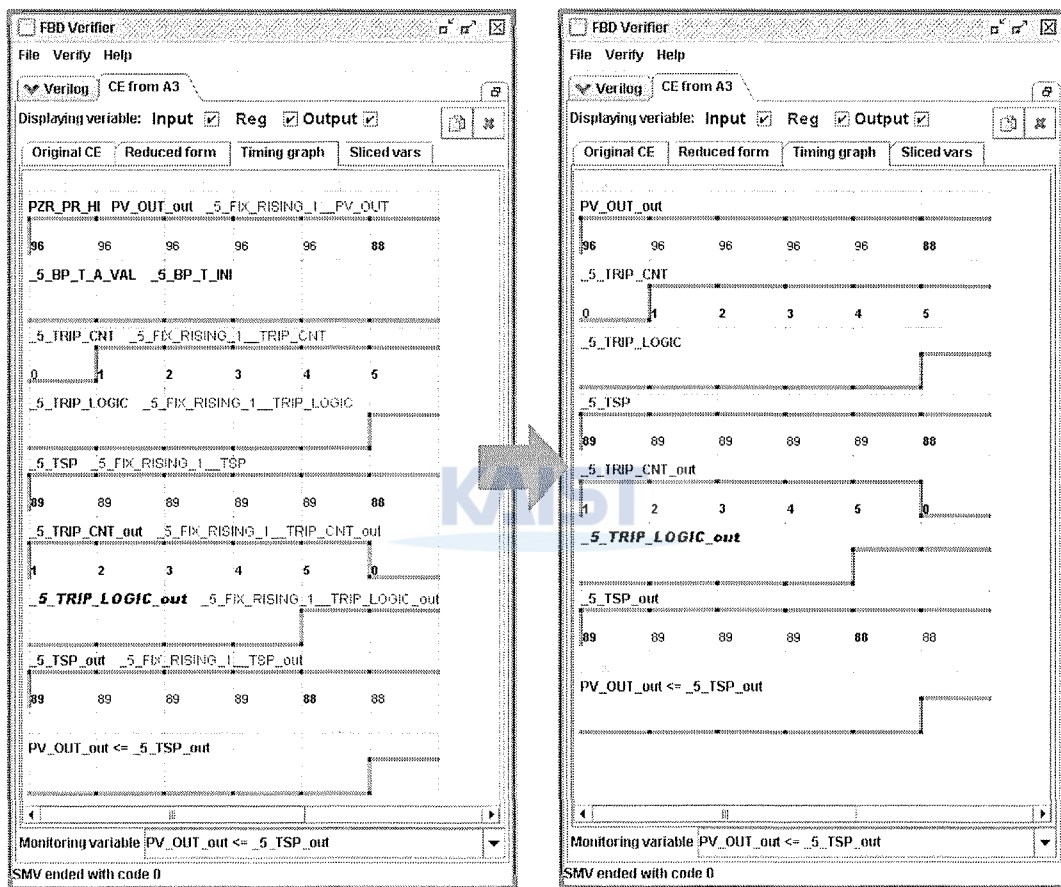


Figure 4.6: FBD2V feature: counterexample sliced by user

5. Related Work

Verilog translation from FBD and verification technique was previously proposed in [2]. It focused on mechanical generation of FBD from the formal specification and equivalence checking using VIS verifier[16] on various versions of FBD program. It originally devised Verilog translation rules in order to use VIS. It also stated the possibility for model checking on translated Verilog program. Main difference of our research is that we focused on model checking and counterexample analysis.

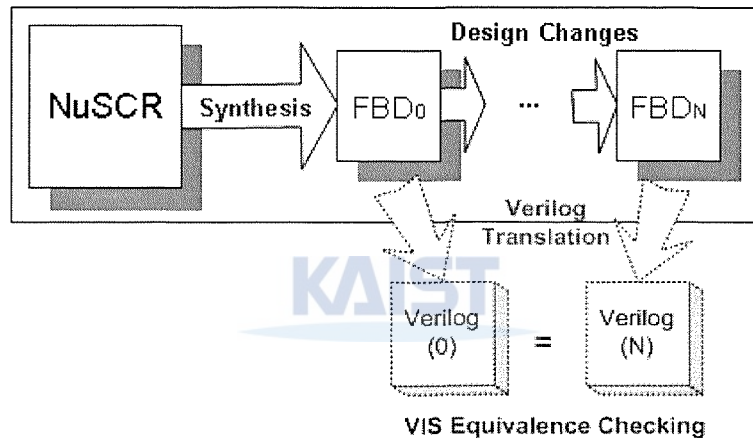


Figure 5.1: FBD generation and equivalence framework by [2]

There are other many Verilog HDL model checkers. CBMC[18] checks Verilog for consistency with ANSI-C program. VCEGAR[19] performs model checking of 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). It can be viewed and analyzed by a wide variety of tools.

6. Conclusion

This paper proposed a framework for formal verification of FBD. We suggested a way to automatically translate Verilog model from FBD program. 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 using model checking. An error residing in a long term of cycles might not be easily found by traditional validation methods, e.g., testing and simulation. The model checking result, *true* guarantees that the program satisfies given properties at any conditions, and counterexamples are key enablers for discovering errors. Second, the tool FBD2V visualizes counterexamples returned by Cadence SMV. Counterexamples with many variables and cycles are hard to be analyzed. FBD2V represents a counterexample in timing graph form which is familiar to hardware engineers. User can declare monitoring variables and slice variables in counterexample to debug the FBD program.

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.

요 약 문

Verilog 변환을 이용한 FBD의 정형검증

원자력 공학 분야에서는 기존에 사용되던 RLL (Relay Ladder Logic) 기반의 아날로그 컨트롤러가 PLC (Programmable Logic Controller) 기반의 디지털 시스템으로 대체되면서 소프트웨어의 안전성에 대한 중요성이 높아지고 있다. KNICS 컨소시엄에서 개발중인 차세대 원자로 APR-1400 RPS 노심보호 시스템은 PLC 개발 언어의 하나인 FBD (Function Block Diagram)으로 SDS (Software Design Specification: 상세설계)가 작성되어 있으며, 이를 대상으로 다양한 V&V 가 이루어지고 있다. 정형기법은 소프트웨어의 안전성을 극대화시키는 방법으로서 주목받고 있으며, 원자력 분야에서도 적용되고 있다. 이 가운데서 모델체킹 기법이 FBD로 작성된 이 시스템을 정형검증하는데 사용된다. 본 연구에서는 FBD 프로그램을 정형검증하기 위한 방법을 제안한다.

먼저 본 연구는 모델체커로 Cadance SMV를 사용하며, 모델체킹에 사용될 속성이 원자력 분야 전문가들에 의해 정해져 있음을 가정한다. 모델체킹이 행해질 대상은 FBD 프로그램으로부터 자동 변환된 Verilog 모델이 된다. Verilog 모델은 본 연구에서 제안된 변환 기법에 의하여 FBD로부터 생성된다. FBD Verifier 도구가 개발되어 Verilog 모델 자동생성, 모델체커 실행, 반례 분석에 이르는 일련의 정형검증 과정을 지원한다. 제안된 방법을 KNICS APR-1400 RPS 시스템에 적용하여 다수의 오류를 발견하였다.

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 portection 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.



감사의 글

논문을 완성하기까지 주위의 모든 분들로부터 수많은 도움을 받았습니다. 먼저 2년간 아낌없는 가르침과 사랑을 주신 차성덕 지도교수님께 깊은 감사를 드립니다. 아울러 소프트웨어공학 연구실의 권용래 교수님, 배두환 교수님께도 많은 것을 배웠습니다. 바쁘신 와중에도 본 논문의 심사를 맡아주신 이윤준 교수님, 황규영 교수님께도 감사를 드립니다.

생활을 함께했던 연구실 가족들께도 감사를 드립니다. 작년에 졸업하신 유준범 선배님, 이렇게 훌륭한 연구주제를 주신 덕분에 석사학위를 받을 수 있었습니다. 감사합니다. 가장 많은 시간 동안 연구와 프로젝트를 함께한 지은경 선배님께는 정말 많은 것을 배웠습니다. 가정에 진정한 행복 깃들기를 기원드립니다. 아버지같이, 큰형님같이 연구실의 기둥이 되어주신 방호정 선배님, 어린 저를 돌봐주셔서 감사합니다. 박사과정을 마무리하신 김태효 선배님 축하드립니다. 연구하는 자세를 배웠습니다. 진솔한 조언을 아끼지 않으신 서정석 선배님 감사합니다. 랩의 크고 작은 일을 맡아 해주신 최윤라 선배님 덕분에 화목한 연구실 생활이 있었습니다. 항공우주연구원의 이종인 선배님, 항상 열심히 생활하는 김상록, 이준섭 후배님께도 많은 도움 받았습니다. 또한 소프트웨어 공학 연구실 그룹의 많은 선배님, 후배님들께도 감사드립니다. 모두 훌륭한 연구의 결실을 성취하시길 바라겠습니다. 하나뿐인 제 소중한 동기 신모범 형께는 각별한 감사를 드립니다. 서로 의지하며 생활해나간 지난 2년을 결코 잊지 못할 것입니다. 함께 사회에 나가는 김수현 김민균 형, 원하시는 바 꼭 이루시길 바랍니다. 김현정, 한아림 누나 진학하셔서 훌륭한 연구 하시리라 믿습니다.

원자력연구소에 계신 여러 박사님들께도 감사의 말씀 드리고 싶습니다. 박기용 박사님 감사합니다. 제가 좀더 열심히 해서 더 도움이 되었다면 하는 아쉬움이 남습니다. 따뜻한 격려를 아끼지 않으신 권기춘 박사님 감사합니다. 원자력 분야에서 큰 업적 이루시길 기원합니다.

KAIST 생활을 함께한 친구들에게도 감사드립니다. 5년이나 같은 방에서 생활한 효준이, 좋은 곳에서 원하는 바 꼭 이루기를 기대합니다. 전자과의 해수에게는 큰 도움 받았습니다. 진학해서도 지금까지와 같이 열심히 살며 행복하기를 바랍니다. 여기 다 쓰지 못하는 대구과학고등학교의 친구들, 동아리 SPARCS의 친구들, 선후배님들께도 모두 감사드립니다.

마지막으로 지금의 저를 있게 해 주신 부모님, 귀여운 동생 승목이에게 이 자리를 빌어 사랑한다고 전하고 싶습니다.

이 력 서

이 름 : 전 승 재

생 년 월 일 : 1983년 10월 30일

주 소 : 대구 수성구 황금동 113 경남타운 8/306

E-mail 주 소 : sjjeon@dependable.kaist.ac.kr

학 력

1999. 3. - 2001. 2. 대구과학고등학교 수료

2001. 3. - 2005. 2. 한국과학기술원 전자전산학과 전산학전공 (B.S.)

2005. 3. - 2007. 2. 한국과학기술원 전자전산학과 전산학전공 (M.S.)

The logo for KAIST (Korea Advanced Institute of Science and Technology) is displayed in a light blue color. It consists of the letters 'KAIST' in a bold, sans-serif font, with a horizontal line underneath the letters.