VERIFICATION OF PLC PROGRAMS WRITTEN IN FBD WITH VIS

JUNBEOM YOO, SUNGDEOK CHA[1*] and EUNKYOUNG JEE[2]
Konkuk University, Division of Computer Science and Engineering
1 Hwayang-dong, Gwangjin-gu, Seoul, 143-701, Republic of Korea
[1] Korea University, Department of Computer Science and Engineering
Anam-dong Seongbuk-Gu, Seoul, 136-701, Republic of Korea (Corresponding Author)
[2] KAIST, Department of Electrical Engineering and Computer Science
373-1 Guseong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea
[*]Corresponding author. E-mail : scha@korea.ac.kr

Received August 5, 2008
Accepted for Publication September 16, 2008

Verification of programmable logic controller (PLC) programs written in IEC 61131-3 function block diagram (FBD) is essential in the transition from the use of traditional relay-based analog systems to PLC-based digital systems. This paper describes effective use of the well-known verification tool VIS for automatic verification of behavioral equivalences between successive FBD revisions. We formally defined FBD semantics as a state-transition system, developed semantic-preserving translation rules from FBD to Verilog programs, implemented a software tool to support the process, and conducted a case study on a subset of FBDs for APR-1400 reactor protection system design.

KEYWORDS : Verification, Equivalence Checking, VIS, Verilog, Function Block Diagram, Programmable Logic Controller, IEC-61131

1. INTRODUCTION

Software safety [1] is an important issue for embedded real-time control systems such as those found in nuclear power plants. When verifying safety-critical software, formal methods [2] play critical roles in demonstrating compliance to regulatory requirements. The Korea Nuclear Instrumentation & Control System R&D Center (KNICS) [3] project[1] used the NuSCR [4] formal specification language and tool-set [5] to formally specify and verify software requirements for reactor protection systems (RPS) for the Advance Power Reactor-1400 (APR-1400) [7]. During the design and implementation phases, programmable logical controllers (PLC) software were written in IEC 61131-3 function block diagram (FBD) [8], and software safety was verified thoroughly. Each release of FBDs becomes official only when authorities have verified the software; two types of formal verification, model checking [6] and equivalence checking, were applied to our FBDs. While the former examined whether or not FBD meets required properties, the latter determined behavioral equivalence between two FBD revisions. Units of equivalence checking can vary from a small module to a whole system, and verification tasks fulfill various needs of FBD programmers and safety engineers. Formal verification contributes to the demonstration of the software safety of PLC programs written in FBD.

This paper proposes how the Verification Interacting with Synthesis (VIS) system [9] can automatically verify FBDs. VIS is widely used in hardware analysis, and with its Verilog [10] front-end, it is also suitable for software analysis. VIS supports computational tree logic (CTL) model checking [11], language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis. Although we explored the possibility of using VIS's sequential equivalence checking and simulation to verify FBD programs for the Advance Power Reactor-1400 (APR-1400) RPS, we chose Cadence Symbolic Model Verifier (SMV) [12] for model checking because VIS's CTL model checking has restrictions when specifying properties [13,14].

To enable VIS's equivalence checking using VIS, we first defined the semantics of FBD as a state transition system and developed rules for translating FBDs into semantically equivalent Verilog. We also implemented

[1] Goal of KNICS consortium project (2001 ~ 2008) is to develop a suite of I&C software for use in the next generation Korean nuclear power plant's advanced power reactor, i.e. APR-1400.

the ***FBD Verifier 1.1*** [13] software tool to automate the translation and then used it on a subset of FBDs for APR-1400's RPS. We found VIS equivalence to be effective.

This paper is organized as follows. Section 2 provides background information on FBD and Verilog in brief. A small translation example from FBD to Verilog is introduced. In Section 3, we formally define the FBD as a state transition system and explain the rules used to translate FBDs into Verilog. Section 4 reports an experiment conducted on a subset of FBDs for APR-1400's RPS with VIS equivalence checking. Section 5 discusses related work on PLC program verification, and Section 6 concludes.

2. BACKGROUND

2.1 FBD Programming

The IEC 61131-3 [8] standard includes five PLC programming languages: Structured Text (ST), Function Block Diagram (FBD), Ladder Diagram (LD), Instruction List (IL), and Sequential Function Chart (SFC). FBD's graphical notations and support for networks of software blocks "wired" together in a manner similar to circuit diagrams has lead to its widespread use. Each function block is depicted as a rectangle and is connected to other input/output variables. Among the 10 function block groups mentioned in IEC 61131-3, five that are pertinent

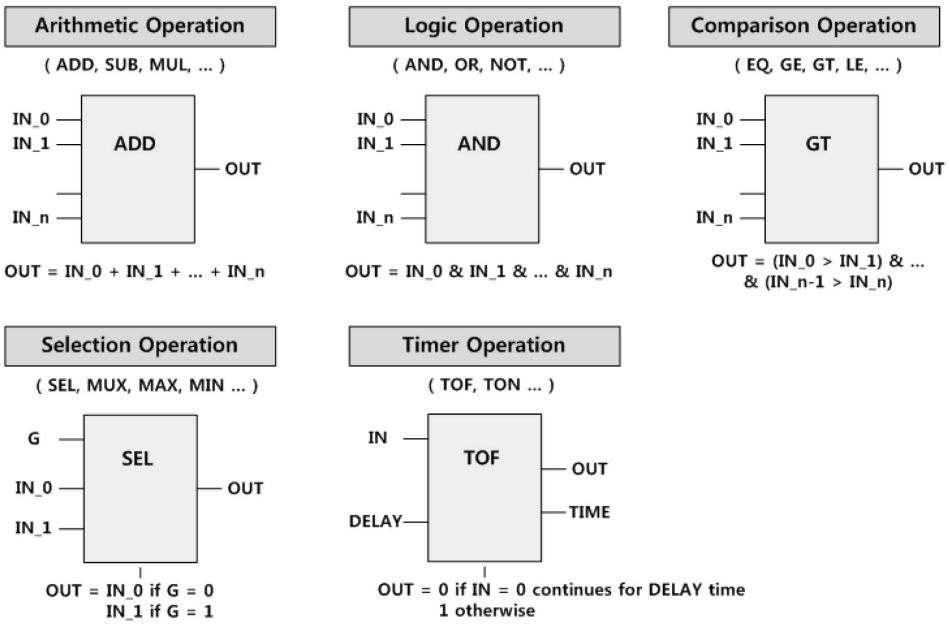


Fig. 1. Function Blocks Defined in IEC 61131-3

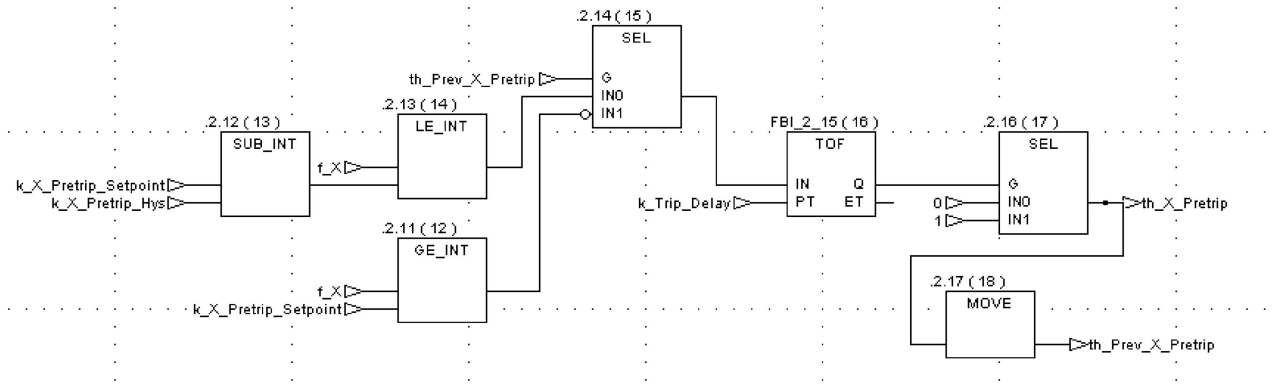


Fig. 2. An FBD for *th_X_Pretrip* Logic

to our discussion are illustrated in Fig. 1. The behavior of a function block is intuitive as their names imply: ADD, AND, SEL, etc.

A portion of FBD for *fixed set-point rising trip logic* in an RPS BP (Bistable Processor) for APR-1400 is shown in Fig. 2. It creates a warning signal, *th_X_Pretrip*, when the trip (e.g. reactor shutdown) condition remains true for *k_Trip_Delay* units of time as implemented in the TOF function block. The number in parenthesis above each function block denotes execution order. The output *th_Prev_X_Pretrip* from MOVE stores the current value of *th_X_Pretrip* for use in the next execution cycle. The TOF function block also maintains internal variables to store timing information.

2.2 Verilog Programming

Verilog is one of the most common hardware description languages (HDLs) used by integrated circuit (IC) designers. Verilog's increasing use in the specification of software logic for process control systems is clear when one considers that designs described in Verilog are technology independent, easy to develop and debug, and are considered more readable than schematics. Verilog has several variable types. A *wire*, similar to a physical wire in a circuit, is used to connect software development modules. *Wires* do not store their values. They are driven by *continuous assignment* statements or by connecting them to other module's outputs. Conversely, *regs*, used in *procedural assignment* blocks beginning with *always*, represent data objects that hold their value between execution cycles.

Fig. 3 shows a Verilog program translated from the FBD from Fig. 2 according to the translation rules described in Section 3. They both show the same behavior as our analyses and experiments verified. There are two inputs (i.e. *f_X* and *th_Prev_X_Pretrip*) and two outputs (i.e. *th_X_Pretrip* and *th_Prev_X_Pretrip*). As input prefixes "*k_*" indicate constants, they are not considered input variables. *Th_Prev_X_Pretrip* is used as both input and output. Since it stores the value of *th_X_Pretrip* using the MOVE function block in FBD, we defined it as a *reg* variable in lines (8) and (32). FBD's output, *th_X_Pretrip*, is produced in the *assign* statements (12)~(18) by composing several function blocks in FBD. It also uses the *timer*

```
(1)  typedef enum {T0, T1, T2, T3, T4, T5} timer_state;
(2)  'define k_Pretrip_Setpoint 30;
(3)  'define k_X_Pretrip_Hys 10;

(4)  module th_X_Pretrip(clk, f_X, th_X_Pretrip);
(5)    input clk;
(6)    input [0:6] f_X;
(7)    output th_X_Pretrip;

(8)    reg th_Prev_X_Pretrip;
(9)    timer_state reg timer;

(10)   initial th_Prev_X_Pretrip = 1;
(11)   initial timer = T0;

(12)   assign th_X_Pretrip =
(13)     (th_Prev_X_Pretrip == 0 && f_X <= 'k_Pretrip_Setpoint - 'k_X_Pretrip_Hys)?1:
(14)     (th_Prev_X_Pretrip == 0 && f_X > 'k_Pretrip_Setpoint - 'k_X_Pretrip_Hys && timer == T5)?0:
(15)     (th_Prev_X_Pretrip == 0 && f_X > 'k_Pretrip_Setpoint - 'k_X_Pretrip_Hys && timer != T5)?1:
(16)     (th_Prev_X_Pretrip == 1 && f_X < 'k_Pretrip_Setpoint)?1:
(17)     (th_Prev_X_Pretrip == 1 && f_X >= 'k_Pretrip_Setpoint && timer == T5)?0:
(18)     (th_Prev_X_Pretrip == 1 && f_X >= 'k_Pretrip_Setpoint && timer != T5)?1:0;

(19)   always @(posedge clk) begin
(20)       if(f_X >= 'k_Pretrip_Setpoint) begin
(21)           case (timer)
(22)                T0: timer = T1;
(23)                T1: timer = T2;
(24)                T2: timer = T3;
(25)                T3: timer = T4;
(26)                T4: timer = T5;
(27)                T5: timer = T5;
(28)           endcase
(29)       else
(30)           timer = T0;
(31)       end

(32)       th_Prev_X_Pretrip = th_X_Pretrip;

(33)   end

(44) endmodule
```

Fig. 3. A Verilog Program Translated from the FBD in Fig.2

variable to emulate a TOF function block, which we emulate with *procedural* assignments using *always* statements (19)~(31). We restricted the number of TOF internal states to six in this example as defined in (1). In addition, we used the *clk* variable, reserved for simulation purposes in VIS, to simulate cyclic executions of the PLC. Thus, the Verilog module defined in Fig. 3 is executed every *clk*-usually 15 to 50 ms.

3. TRANSLATION FROM FBD INTO VERILOG

IEC 61131-3 FBD is a network of function blocks executed sequentially according to a (their) predefined order. In this section, we define the FBD programming language as a state transition system and propose FBD-Verilog translation rules in a bottom-up manner. Section 3.1 explains function block translation, which is a unit of FBDs, and Section 3.2 explains component FBD translation. The translation of system FBD is introduced in Section 3.3. Unlike "primitive" function blocks whose translation rules are straightforward, some FBD features cannot be mapped directly. Such issues are described in subsection 3.4.

3.1 Function Block Translation

A **function block** is defined as a tuple composed of Name, input ports IP, output ports OP, and its behavior description BD as defined in Def. 1. IP and OP are the official terms used in IEC 61131-3. The behavior of a function block is defined as a set of predicates and assignments on input and output ports, respectively. A function block is a function from a set of input variables, which are assigned to input ports of the function block, to (usually) one output variable, which is assigned to one output port.

```
function [0:6] SUB_INT;   // function SUB_INT definition
  input [0:6] in1;        // Two inputs of 7 bits
  input [0:6] in2;
  begin
    SUB INT = in1 - in2;
  end
Endfunction


function SEL;             // function SEL definition
  Input in1, in2, in3;    // Two input and one selection
  begin
    SEL = (in1 == 1)?in3:in2;
  end
endfunction
```

Fig. 4. Verilog Function Definitions for SUB and SEL Function Blocks

Definition 1 (Function Block) A function block is defined as a tuple FB = < Name, IP, OP, BD >, where

- Name: a name of function block
- IP: a set of input ports, $\{ip_1, ip_2, \ldots, ip_n\}$
- OP: a set of output ports, $\{op_1\}$
- BD: behavioral description $\sum(p_{FB}, a_{FB})$, where
  - p_{FB} : a predicate on IP
  - a_{FB} : assignments on OP

Let $V_{FB\text{-}I} = \{v_1, v_2, \ldots, v_n\}$ be a set of function block input variables which has n input ports, and $V_{FB\text{-}O} = \{v_o\}$ be a set of function block output variables. If we define Ii as a set of input domains of the input variable v_i ($1<=i<=n$) and $I_{FB} = I_1 \times I_2 \times \ldots \times I_n$, and also $O_{FB} = O_o$ in the same way, then a function block is defined as follows: $f_{FB}: I_{FB} \rightarrow O_{FB}$

The five rules shown below define how to translate a function block into an equivalent Verilog function. The translation rules are straightforward because there exists little semantic gap between the two. They both receive inputs, process data, and emit outputs. The output port type (i.e. Boolean, integer, or bit vector) determines the corresponding Verilog function type as **Rule 1**. Explicit type conversion might be needed when the output type of a function block is not supported by Verilog, i.e. real type variables. Similarly, each input and its type is declared as described in **Rule 2**. Function behavior is translated next as shown in **Rules 3** and **4**. As an example, Fig. 4 illustrates how the SUB and SEL function blocks are translated into Verilog.

1. for each *FB = < Name, IP, OP, BD >*:

   function [type] *Name* ;

2. for each $ip_i \in IP$:

   input [type] ip_i;

3. for a *BD* of *logical, arithmetic operation* $\oplus$:

   begin

   $Name = ip_1 \oplus ip_2 \oplus \ldots \oplus ip_n$;

   end

4. for a *BD* of *selection operation*:

   (provided $IP = \{k, ip_0, ip_1, \ldots, ip_n\}$)

   begin

   Name = (*k* == 0) ? ip_0 :

   (*k* == 1) ? ip_1 :

   …

   (k == n) ? ip_n : *default* ;

   end

5. finishing remark of *function* declaration

   endfunction

Unlike simple function blocks, translation rules are more complex when it comes to timer operations. One must use *reg* type variables to track internal information. In addition, a timer's internal variable has to have discrete bounds. As a Verilog function cannot hold procedural assignments that treat *reg* variables, we use Verilog module feature instead.

3.2 Component FBD Translation

A **Component FBD** is a logical block of independent function blocks to which a number of function blocks are interconnected to generate meaningful outputs. Fig. 2 is an example of a component FBD for *th_X_Pretrip* logic. A component FBD is defined as a tuple composed of a set of function block FBs, a set of transitions T between the function blocks, a set of input ports I, and a set of output ports O. Inputs to a component FBD come from other component FBDs or system input variables.

Definition 2 (Component FBD) Function block diagram is defined as a tuple Component_FBD = < FBs, T, I, O >, where

- FBs: a set of function block FBs
- T:
  - a set of transitions ($FB_i.OP_m$, $FB_j.IP_n$) between function blocks FB_i and FB_j in FBs (provided that $i \neq j$, and $FB_j.IP_n$ means n^{th} input port of function block FB_j)
  - $\forall (FB_i.OP_m,\ FB_j.IP_n) \in T,\ FB_i$ has sequential execution precedence on FB_j
- I: a set of *FB.IP* which do not appear in T and are assigned by $V_{comp\_FBD\text{-}I}$
- O: a set of *FB.OP* which do not appear in T and are assigned by $V_{comp\_FBD\text{-}O}$

The semantics of a component FBD are defined as a function from a set of input variables to output variables. Let $V_{Comp\_FBD\text{-}I} = \{v_{ci1}, v_{ci2}, ..., v_{cim}\}$ denote a set of input variables entering the FBD and $V_{Comp\_FBD\text{-}O} = \{v_{co1}, v_{co2}, ..., v_{con}\}$ a set of output variables leaving the FBD. Variables in $V_{Comp\_FBD\text{-}I}$ are assigned to input ports I, and those in $V_{Comp\_FBD\text{-}O}$ are assigned to output ports O. If we define I_ω as a set of input domains of input variable $v_{ci\omega}$ ($1<=\omega<=m$) and $I_{Comp\_FBD} = I_1 \times I_2 \times ... \times I_m$, and $O_{Comp\_FBD} = O_1 \times O_2 \times ... \times O_n$ in the same way, a component FBD is defined as the following function: $f_{Component\_FBD}: I_{Comp\_FBD} \rightarrow O_{Comp\_FBD}$

When translating a component FBD, we use the Verilog module construct and assume that all function blocks have already been translated into Verilog functions. A module invokes functions or other modules according to their predefined execution order in the FBD program. First, the component FBD's name and ports are declared in **Rules 6** through **8**. **Rule 9** declares and initializes *reg* type variables. If an output variable is also used as input, it is declared as *reg* type as its value is to be used in the next cycle. Variable *th_Prev_X_Pretrip* in Fig. 2 is such an example. Verilog provides a number of features, i.e. for loops, while loops, and case statements, to support succinct description of complex behavior. In **Rule 10**, [*Verilog_function_calls*] calls every Verilog function according to its execution order to generate outputs of the component FBD. Every function block is separately translated as a Verilog function and included in the definition of module for the component FBD. A Verilog definition ends with the reserved word *endmodule* in **Rule 12**.

6. for each *Component_FBD* = <*FBs, T, I, O* >:
   module [Name_of_Module] ([input_variables_list], [output_variable_list]) ;
7. for each *FB.IP* $\in$ *I* and its corresponding $v_i \in V_{Comp\_FBD\text{-}I}$:
   input [type] v_i;
8. for each *FB.OP* $\in$ *O* and its corresponding $v_o \in V_{Comp\_FBD\text{-}O}$:
   output [type] v_o;
9. for each stored variable v_{reg} :
   reg [type] v_{reg} :
   initial v_{reg} = [initial_type] ;
   always @(posedge clk) begin
   [*procedural_assignment_for_*v_{reg}] ;
   end
10. for each output variable $v_o \in V_{Comp\_FBD\text{-}O}$:
    assign v_o = [*Verilog_function_calls*] ;
11. for function blocks $\in$ *FBs*, include function declarations:
    function … ;
    …
    endfunction
12. finishing remark of *module* declaration
    endmodule

To produce optimal Verilog code, the use of Verilog functions must be avoided when equivalent expressions exist. For example, GE_INT in Fig. 2 can be translated into the built-in "*f_X >= k_X_Pretrip_Setpoint*" expression or a user-defined *function GE_INT*; the former translation is preferred. Fig. 3 shows how Rules 1 through 12 are applied to the FBD shown in Fig. 2. Complete technical details are available elsewhere [15].

3.3 System FBD Translation

The whole FBD software system is composed of a number of component FBDs and their interconnections. A **System FBD** defines the whole software system as a tuple composed of component FBDs Component_FBD, a set of transitions T between component FBDs, a set of input ports I, and a set of output ports O as defined in Def. 3.

Definition 3 (System FBD) System FBD is defined as a tuple System_FBD = < FBDs, T, I, O >, where

- FBDs: a set of component FBDs component_FBDs
- T:

- a set of transition ($FBD_i.O_m$, $FBD_j.I_n$) between FBD_i and FBD_j in FBDs (provided that $i \neq j$ and $FBD_j.I_n$ is an n-th input port of FBD_j.
- $\forall(FBD_i.O_m, FBD_j.I_n) \in T$, FBD_i has a sequential execution precedence on FBD_j
- I: a set of *FBD.I* which do not appear in T and are assigned by $V_{Sys\_FBD\text{-}I}$
- O: a set of *FBD.O* which do not appear in T and re assigned by $V_{Sys\_FBD\text{-}O}$

Similarly, a system FBD is defined as a function from a set of system input variables $V_{Sys\_FBD\text{-}I} = \{v_{si1}, v_{si2}, ..., v_{sim}\}$ to a set of system output variables $V_{Sys\_FBD\text{-}O} = \{v_{so1}, v_{so2}, ..., v_{son}\}$. If we define I_{sv} as a set of input domains of the input variable $v_{siv}(1 <= v <= m)$ and $I_{Sys\_FBD} = I_{s1} \times I_{s2} \times ... \times I_{sm}$, and also $O_{Sys\_FBD} = O_{s1} \times O_{s2} \times ... \times O_{sn}$ in the same way, then the system FBD can be regarded as a function: f_{System_FBD}: $I_{Sys\_FBD} \rightarrow O_{Sys\_FBD}$

A system FBD contains a number of component FBDs and their sequential interconnections. While translation rules for system FBDs look similar to the rules of component FBDs, it uses [*Verilog_module_instantiations_for_*v_o] instead of [*Verilog_function_calls*]. Verilog modules are instantiated and called according to their execution order with outputs communicated. For example, the Verilog program for the *g_LOG_PWR* module, a larger system depicted in Fig. 5, instantiates 6 modules, *M_f_X_Generation* ~ *M_th_X_Trip* into SIM_PURPOSE ~ EE, respectively. The Verilog module *th_X_Pretrip* in Fig. 3 corresponds to a module instantiation "*M_th_X_Pretrip EE*" in Fig. 5.

13. for a *System_FBD* = <*FBDs, T, I, O* >:
    module [Name_of_Module] ([input_variables_list], [output_variable_list]) ;
14. for each *FBD.I* $\in$ *I* and its corresponding $v_i \in V_{Sys\_FBD\text{-}I}$:
    input [type] v_i ;
15. for each *FBD.O* $\in$ *O* and its corresponding $v_o \in V_{Sys\_FBD\text{-}O}$:
    output [type] v_o;
16. for each intermediately used wire variable v_{wire} :
    wire [type] v_{wire} ;
17. for each stored variable v_{reg} :
    reg [type] v_{reg} :
    initial v_{reg} = [initial_type] ;
    always @(posedge clk) begin
    [*procedural_assignment_for_*v_{reg}] ;
    end
18. for each output variable $v_o \in V_{Sys\_FBD\text{-}O}$:
    [*Verilog_module_instantiations_for_*v_0] ;
19. finishing remark of *module* declaration
    endmodule

```
module g_LOG_PWR (clk, f_X_Raw, f_X_Ob_Ini, f_Channel_Error,f_Module_Error,
    th_X_Trip, h_X_OB_Sta, f_X_OB_Perm, th_X_Pretrip);
  input clk;
  input [0:6] f_X_Raw;
  input f_X_Ob_Ini;
  input f_Channel_Error;
  input f_Module_Error;

  output th_X_Trip;
  output h_X_OB_Sta;
  output f_X_OB_Perm;
  output th_X_Pretrip;

  wire [0:6] f_X;
  wire f_X_Valid;

  M_f_X_Generation  SIM_PURPOSE(clk, f_X_Raw, f_X);
  M_f_X_Valid  AA(f_X, f_X_Valid);
  M_f_X_OB_Perm  BB(f_X, f_X_OB_Perm);
  M_h_X_OB_Sta  CC(f_X_OB_Perm, h_X_OB_Sta);
  M_th_X_Pretrip  DD(clk, f_X, th_X_Pretrip);
  M_th_X_Trip  EE(clk, f_X, f_X_Valid, h_X_OB_Sta, f_Module_Error,f_Channel_Error, th_X_Trip);
endmodule
```

Fig. 5. A Verilog Program for *g_LOG_PWR System* FBD

It is important that all component FBDs should be *a priori* defined as Verilog modules and that their transition relations be defined in T of System_FBD. Verilog module instantiations should precisely reflect these transition relations. One can translate the whole FBD program into an equivalent Verilog program hierarchically in a similar manner.

3.4 Practical Considerations on Translation Rules

When applying the above rules to large and complex real-world situations, the following guidelines are useful. First, one must distinguish "intermediate" FBD outputs from "externally visible" FBD outputs. That is, outputs of FBD subsystems must not be mapped as output variables. Rather, they must be declared as "internal" *reg* variables in the higher-level Verilog modules. Variable *th_Prev_X_Pretrip* in Fig. 2 is such an example. Correctness on data dependency and proper ordering among the outputs of FBD subsystems are critical in translation.

Second, the translation of timer blocks (e.g. TOF and TON) requires synchronization between the global system clock, *clk*, and multiple local clocks. Fig. 6 describes a template for translating a TOF function block, which has IN and DELAY as inputs, and OUT and TIME as outputs. IN is an input Boolean variable, and DELAY is a variable specifying time delay. OUT is an output Boolean variable, and TIME represents the elapsed time of its internal timer. Output value OUT is 0 if input IN remained 0 during DELAY time periods since input IN changed from 1 to 0. Otherwise, the output is 1. The TOF described in Fig. 6 has up to 3 bits delay, or 2^3=8 clock time. Output TIME is excluded in the Verilog code because it is used to monitor elapsed time of the local timer for simulation and debugging purposes only. The local clock of the timer is synchronized with the global clock *clk* using the "***always*** @(*posedge clk*) ***begin***" statement. Each timer block can be declared as a separate Verilog module, and the behavior of multiple timers can be unfolded within the component FBD's definition as shown in Fig. 3.

Third, one must be careful in deciding the number of bits used to represent timer delay values. Larger values exponentially increase the number of states to be explored. In the worst case, state explosion may occur in SMV model checking or VIS equivalence checking. Other blocks (e.g. Bistable or Counters) share the same issue.

```
module TOF (clk, IN, DELAY, OUT);

 input clk, IN
 input [0:2] DELAY;
 output OUT;

 reg [0:2] timer;
 initial timer = 3'b000;

 assign OUT = ! IN && (timer == DELAY)?0:1;

 always @(posedge clk) begin
   if(! IN) begin
     if(timer < DELAY) timer = timer + 3'b001;
     else
       timer = DELAY;
   end
   else
       timer = 3'b000;
 end
endmodule
```

Fig. 6. A Verilog Program for TOF Timer Function Block

4. VIS EQUIVALENCE CHECKING

We applied the proposed translation techniques to a subset of FBDs [16] for APR-1400's RPS BP currently under development in Korea and performed VIS equivalence checking on them. As the safety of APR-1400's RPS must be rigorously demonstrated, regulatory bodies strongly recommend that formal verification techniques be used. While model checking verifies whether or not FBD meets required properties, it is inadequate for verifying the behavioral equivalence between two FBD versions. On the other hand, VIS equivalence checking is particularly useful for checking if FBD design optimizations do not introduce errors.

In order to evaluate the effectiveness of the proposed approach, we conducted a case study using a subset of the preliminary version of bistable processor (BP) design. BP, as a part of a reactor protection system, is large and complex in that its design specifications, Rev. 02 released in 2006, consist of 1,355 function blocks and 1,038 variables; when translated using ***FBD Verifier 1.1***, the Verilog program was 7,862 lines long. Section 4.1 introduces an overview of the VIS equivalence checking process, and the detailed experimental results are explained in Section 4.2.

4.1 An Overview of Verification Process

Fig. 7 shows an overview of the VIS equivalence checking process. FBD designs are programmed with the pSET [17] commercial engineering tool developed by the PLC vendor POSCON. The designs are subsequently compiled into executable code. *FBD Verifier 1.1* translates the design into equivalent Verilog programs in "*.v*" format. As the VIS verification system has no graphical user interface, we executed the VIS in a ***Cygwin*** environment where equivalence checking was performed. A program (***vl2mv*** [18]) in the VIS verification system translates Verilog programs from the "*.v*" into the "*.mv*" format, which VIS can read and analyze. If any evidence of inequivalence is found, VIS generates a counterexample illustrating how changes on values of various variables lead to different behavior.

4.2 Experimental Result

We performed the proposed automatic verification on

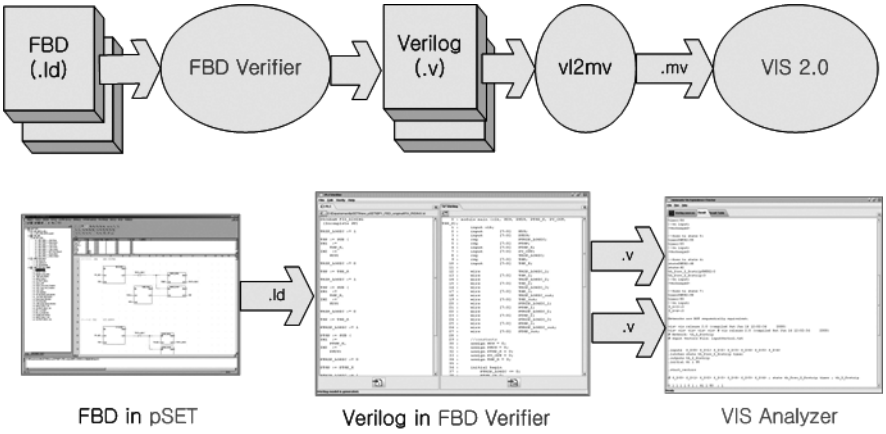


Fig. 7. An Overview of VIS Equivalence Checking

a subset of FBDs for APR 1400's RPS BP. The FBD is depicted in Fig. 2. It is an important part of BP logics, and was excerpted from a preliminary release of FBDs [16] for the purpose of this experiment. A different version of FBD is shown in Fig. 8. It should have the same behavior as the FBD in Fig. 2 because it was mechanically synthesized from the same NuSCR requirements specification [19] using a synthesis technique previously reported [20]. Synthesis version allows FBD engineers to validate manually developed and optimized FBD programs even if the synthesized FBDs do not become part of an official release.

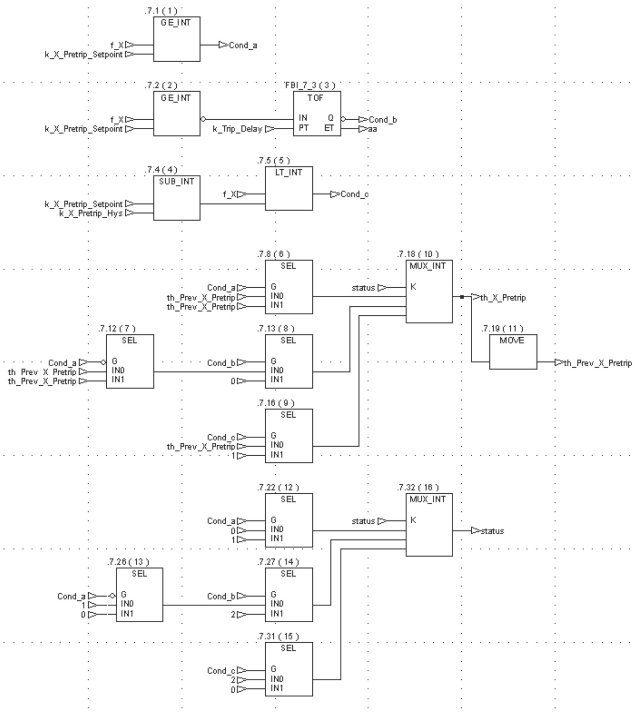


Fig. 8. A Mechanically Synthesized FBD for *th_X_Pretrip* Logic

When VIS equivalence checking was performed, to our surprise, VIS determined the behaviors of the two FBD programs to be different and generated a seven-step counterexample (Fig. 9)[2]. For example, on transition from state 0 to 1, the value of "timer" changed from "T0" to "T1." Such changes are caused by the input *f_X* of "1011110." According to the counter-example, the inequivalence occurred when the pretrip fired (*th_Prev_X_Pretrip* = 0 in state 6), and then the pretrip condition was released in the next state 7 on input "0010110." As the VIS counter-example does not show different output values explicitly in the final state, we must use the simulation facility of VIS to fully investigate the cause for the in-equivalence and fix the errors.

Although the details are beyond the scope of this paper, the SEL function block numbered (15) in Fig. 2 has G input wired to *th_Prev_X_Pretrip*. It means that if the value is 1 then it is currently in a normal state, otherwise it is in a pretrip state. Although Timer function blocks should not be used when the system is in a pretrip state (i.e., *th_Prev_X_Pretrip* = 0), TOF timer block numbered (16) is executed after the SEL function block. Such redundant use of the TOF timer gave rise to different behavior, and domain engineers fixed the error by separating the TOF block so that it is used only when the system is in the normal state. Fig. 10 shows the modified FBD in which the computation on the TOF block takes prior to that of the SEL block. The VIS equivalence checking result was also a success (Fig. 11). The modified FBD was used in later releases.

We also applied VIS equivalence checking to the

[2] We edited and visualized it to aid understanding.

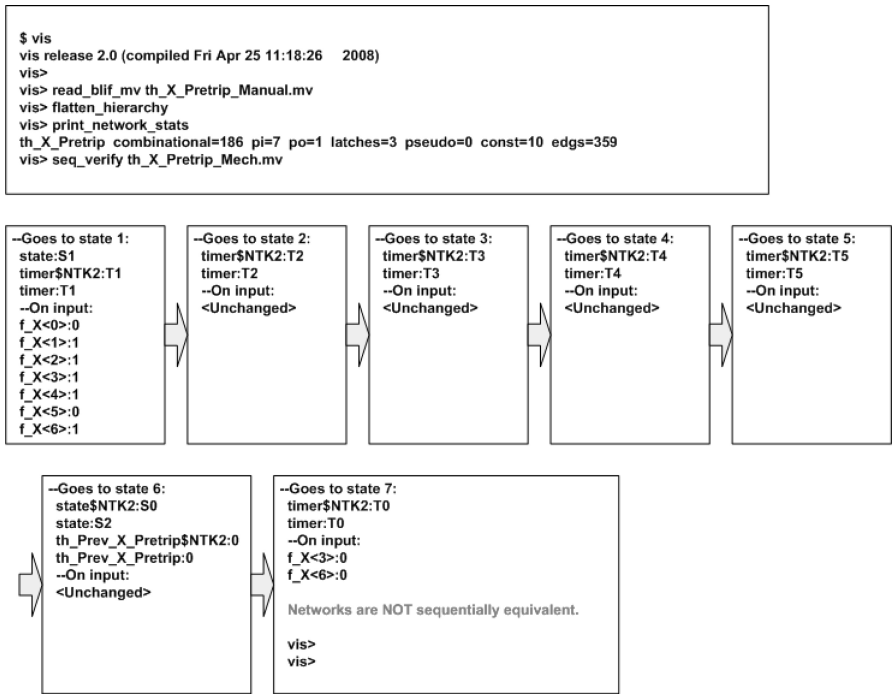


Fig. 9. A Counter-Example of VIS Equivalence Checking

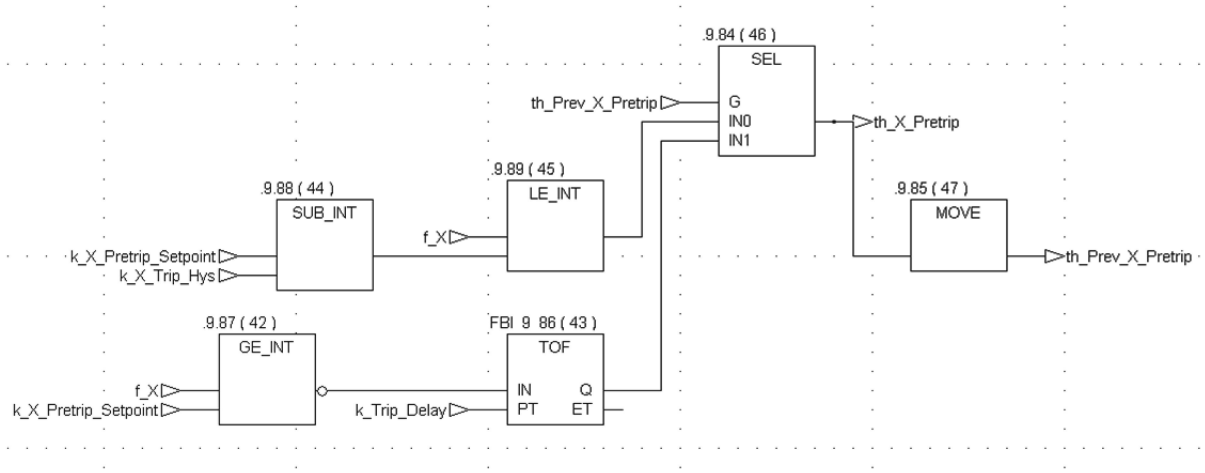


Fig. 10. A Modified FBD for *th_X_Pretrip*

FBDs for "*manual reset variable set-point trip logic*" of the APR-1400 RPS BP, which is more complex than the "*fixed set-point rising trip logic*" used above. We found several critical errors in both the manually developed and the synthesized FBD versions (Table 1). In addition to syntactic and trivial mistakes made by FBD engineers, we also detected several logical errors, which included some critical discrepancies between the two FBDs. Such errors could not easily be solved by changing the order of the function blocks or by replacing them. We even found logical errors in the NuSCR requirements specification [19]. All errors were fixed, and the requirements and design specification documents were updated.

As mentioned earlier, analyzing the results of VIS equivalence checking counterexamples is often time prohibitive for FBD engineers and verifiers. VIS is executed in ***Cygwin*** or ***Linux*** environments in text mode, and the counterexample generated contains a lot of value information at the bit-level. We also had to rely on VIS's simulation to understand the output of FBDs. Even though VIS's equivalence checking is an efficient and useful automatic verification technique, the above obstacles render its widespread use in developing FBD programs difficult. We are currently developing a tool to assist visual

Table 1. VIS Equivalence Checking Result for APR-1400 RPS BP

Trip Logic	Error Type	Synthesized FBD (Num. of Errors)	Original FBD (Num. of Errors)
Fixed Set-Point Rising Trip without Operating Bypass	Syntactic	0	0
	Logical	0	1
Manual Reset Variable Set-Point Trip without Operating Bypass	Syntactic	0	3
	Logical	6	2

```
$ vis
vis release 2.0 (compiled Fri Apr 25 11:18:26     2008)
vis> read_blif_mv th_X_Pretrip_Mech.mv
vis> flatten_hierarchy
vis> seq_verify th_X_Pretrip_Manual_Modified.mv
Networks are sequentially equivalent.

vis>
vis>
```

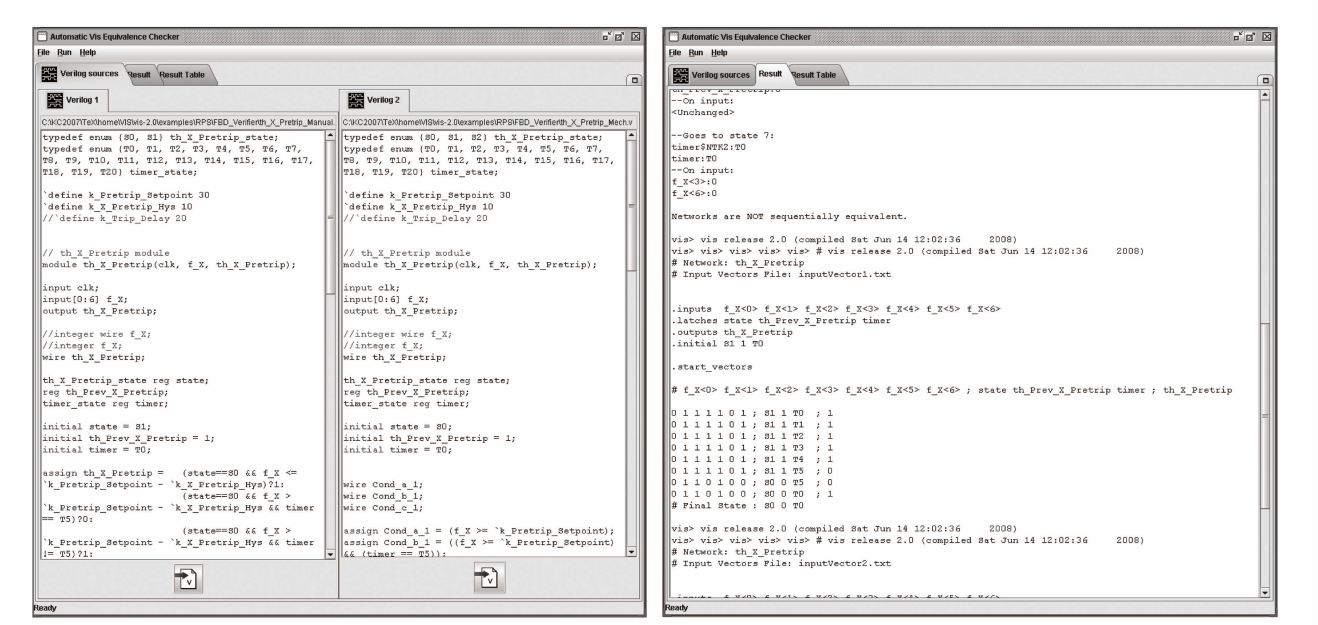
Fig. 11. VIS Verification Result after the Modification

equivalence analyses of counterexamples. Fig. 12 shows an interface design of a ***VIS Analyzer 0.8*** prototype. Fig. 12a shows two Verilog programs read, and Fig. 12b is a result of the automatic execution of several VIS operations, i.e. ***vl2mv, seq_verify***, and simulate.

5. RELATED WORK

Many approaches, including a PLCTOOLS tool-set [22], for the formalization of PLC programs written in LD, ST, and FBD indented for formal verification, validation, and simulation exist in the literature [21]. FBD programs are modeled with high-level timed petri nets (HLTPN), and those nets are then used for design validation and code generation. ***MATLAB/SIMULINK*** provides a means for specifying and simulating plants. While *PLCTOOLS* focuses on designing, simulation, and PLC code generation, it does not support formal verification such as VIS equivalence checking.

Higher order logic (HOL) has been used to model requirements specifications [23]. There are no restrictions on data types in this approach since function blocks are modeled as relations on HOL streams. The Verilog we use has several restrictions on data types, i.e. Verilog has no real number type variables. It treats time implicitly,



(a) Two Verilog programs read (b) VIS equivalence checking and simulation result

Fig. 12. A *VIS Analyzer 0.8 Screen-Dump*

which is contrary to the Verilog and VIS verification systems. However, proofs are done with the help of the Isabelle/HOL [24] theorem prover. Compared to automatic verification using VIS, its cost is prohibitive.

In IEC 61499 [25] interactions were defined between controllers and overall systems (plants) using FBDs; they were formalized [26] with a single-net systems (SNS) [27] model. The controller code was defined in FBD format, and the overall system was organized in IEC 61499 function blocks. In this approach, the complete structure is automatically translated into an SNS model using the Verification Environment for Distributed Environment (***VEDA***) tool. For the combined plant-controller model, model checking was performed using Singal/Event System Analyzer (***SESA***) [28]. While VIS equivalence checking verifies FBD programs from a unit module to a whole system, it focuses on the interactions between IEC 61499 function blocks, which correspond to subsystems and not detailed IEC 61131-3 function block diagrams.

In case of the automatic formal verification model checking of Verilog programs, there are several approaches using different model checkers. We have used the ***SMV*** model checker as previously implemented and experimented upon [13, 14]. As explained in Section 1, VIS's CTL model checking has some rigorous restrictions. We used ***Cadence SMV***'s LTL model checking to avoid them. ***CBMC*** [29] checks Verilog for consistency with ANSI-C programs. ***VCEGAR*** [30] performs model checking on Verilog using a counter-example guided abstraction refinement framework. However, these approaches have not yet been applied to the development of safety-critical software in industry except for the *SMV* approach.

6. CONCLUSION AND FUTURE WORK

This paper described effective use of VIS, a well-known verification tool, for automated software verification of PLC programs written in FBD. We formally defined FBD semantics as a state-transition system, developed translation rules from FBD to Verilog programs, implemented FBD Verifier 1.1 software tool to automate the translation, and performed VIS verification-equivalence checking on a subset of FBDs currently under development for an APR-1400 nuclear power reactor. The case study demonstrated that behavioral equivalence checking using VIS is an effective and useful verification technique that can easily be used in developing PLC programs.

As we perform VIS equivalence checking in *Cygwin* or *Linux* environment in text mode, automating the verification process and visualizing analysis results including VIS simulation would promote its applicability and usability remarkably. We also introduced a prototype tool design. We are planning on developing a tool suite supporting development, verification, and safety analysis throughout entire lifecycle phases from requirements specification to implementation.

ACKNOWLEDGEMENTS

The Korean Research Foundation Grant funded by the Korean Government (KRF-2008-331-D00524) and a Korea University Grant supported this research. This research was also partially supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2008-(C1090-0801-0032)).

REFERENCES

[1] N.G. Leveson, SAFEWARE, *System safety and Computers*, Addison Wesley, 1995.
[2] D.A. Peled, *SOFTWARE RELIABILITY METHODS*, Springer, 2001.
[3] KNICS, Korea Nuclear Instrumentation & Control System R&D Center, http://www.knics.re.kr/.
[4] J. Yoo, T. Kim, S. Cha, J.S. Lee, and H.S. Son, "A formal software requirements specification method for digital nuclear plants protection systems," *Journal of Systems and Software*, **74**, **1**, pp.73-83, 2005.
[5] NuSRS, http://dependable.kaist.ac.kr/~nusrs.
[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Programming Languages and Systems*, **8**, **2**, pp.244–263, 1986.
[7] J. Cho, J. Yoo, and S. Cha, "NuEditor - a tool suite for specification and verification of NuSCR," *SERA2004:Second ACIS International Conference on Software Engineering Research*, Management and Applications, pp.298–304, 2004.
[8] IEC, *International standard for programmable controllers: Programming languages 61131- Part 3*, 1993.
[9] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.T. Cheng, S.A. Edwards, S.P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa, "VIS : A system for verification and synthesis," the Eighth International Conference on Computer Aided Verification," *CAV '96*, pp.428–432, 1996.
[10] D.E. Thomas and P.R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.
[11] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 1999.
[12] Cadence SMV, http://www.cadence.com.
[13] S.J. Jeon, "Verification of Function Block Diagram through Verilog Translation," Master's thesis, Korea Advanced Institute of Science and Technology, 2007.
[14] J. Yoo, E. Jee, and S. Cha, "A Verification framework for FBD based software in nuclear power plants," The 15th Asian Pacific Software Engineering Conference (APSEC 2008), pp.385-392, 2008.
[15] J. Yoo, Synthesis of Function Block Diagrams from NuSCR Formal Specification, Ph.D. thesis, Korea Advanced Institute of Science and Technology, 2005.
[16] KAERI, *SDS for reactor protection system*, KNICS-RPS-SDS101 Rev.00 Draft, Sept. 2003.
[17] S. Cho, K. Koo, B. You, T.W. Kim, T. Shim, and J.S. Lee,

"Development of the loader software for PLC programming," *Conference of the Institute of Electronics Engineers of Korea*, pp.959–960, 2007.
[18] S.T. Cheng and R.K. Brayton, "Compiling verilog into automata," Tech. Rep. UCB/ERL M94/37, EECS Department, University of California, Berkeley, 1994.
[19] KAERI, *SRS for Reactor Protection System*, KNICS-RPS-SRS101 REV.00, Feb. 2003.
[20] J. Yoo, S. Cha, C.H. Kim, and D.Y. Song, "Synthesis of FBD-based PLC design from NuSCR formal specification," *Reliability Engineering and System Safety*, **87**, **2**, pp.287-294, 2005.
[21] G. Frey and L. Litz, "Formal moethods in PLC programming," *IEEE Conference in System Man and Cybernetics:SMC 2000*, 2000.
[22] L. Baresi, M. Mandrioli, S. Morasca, and M. Pezz`e, "Plctools: Design, formal verification, and code generation for programmable controllers," *the IEEE Conference on System, Man, and Cybernetics* (*SMC*), Nashville, USA, pp.2437-2442, Oct. 2000.
[23] B.J. Kramer and N. Volker, "A higher dependable computer architecture for safety critical control applications," *Real-Time Systems Journal*, **13**, **3**, pp.237-251, 1997.
[24] T. Nipkow, L.C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, LNCS, vol.2283, Springer, 2002.
[25] IEC, *Function blocks - Part 1: Architecture* (*IEC 61499-1*), 2005.
[26] V. Vyatkin and H.M. Hanisch, "Modeling of IEC 61499 function blocks - a cue to their verification," *XI Workshop on Supervising and Diagnostics of Machining System*, pp.59-68, 2000.
[27] P.H. Starke, "Symmetries of signal-net systems," *Workshop on Concurrency, Specification and Programming*, pp.285-297, 2000.
[28] P.H. Starke and S. Roch, "Tools for formal specification, verification, and validation of requirements," *the 12th Annual Conference on Computer Assurance*, *COMPASS '97*, pp.35-47, 1997.
[29] Bounded Model Checking for ANSI-C, http://www.cs.cmu.edu/~modelcheck/cbmc.
[30] H. Jain, N. Sharygina, D. Kroening, and E. Clarke, "Word level predicate abstraction and refinement for verifying rtl verilog," *42nd Design Automation Conference* (*DAC*), 2005.