http://dx.doi.org/10.5516/NET.04.2012.085

A BEHAVIOR-PRESERVING TRANSLATION FROM FBD DESIGN TO C IMPLEMENTATION FOR REACTOR PROTECTION SYSTEM SOFTWARE

JUNBEOM YOO[1*], EUI-SUB KIM[1], and JANG-SOO LEE[2]
[1] Konkuk University, Division of Computer Science and Engineering
1 Hwayang-dong, Gwangjin-gu, Seoul, 143-701, Republic of Korea
[2] Korea Atomic Energy Research Institute, Man-Machine Interface System Team
989-111 Deadeok-daero Yuseong-gu, Daejeon, 305-353, Republic of Korea
[*]Corresponding author. E-mail : jbyoo@konkuk.ac.kr

Received November 30, 2012
Accepted for Publication February 12, 2013

Software safety for nuclear reactor protection systems (RPSs) is the most important requirement for the obtainment of permission for operation and export from government authorities, which is why it should be managed with well-experienced software development processes. The RPS software is typically modeled with function block diagrams (FBDs) in the design phase, and then mechanically translated into C programs in the implementation phase, which is finally compiled into executable machine codes and loaded on RPS hardware - PLC (Programmable Logic Controller). Whereas C Compilers are fully-verified COTS (Commercial Off-The-Shelf) software, translators from FBDs to C programs are provided by PLC vendors. Long-term experience, experiments and simulations have validated their correctness and function safety. This paper proposes a behavior-preserving translation from FBD design to C implementation for RPS software. It includes two sets of translation algorithms and rules as well as a prototype translator. We used an example of RPS software in a Korean nuclear power plant to demonstrate the correctness and effectiveness of the proposed translation.

KEYWORDS : Behavior-Preserving Translation, Programmable Logic Controller, Translator, Function Block Diagram, C Program

1. INTRODUCTION

Safety [1] is an important property for nuclear power plants in order to obtain permission from government authorities for their operation and possible export of power plant construction technology. As the nuclear reactor protection system (RPS) makes decisions for emergent reactor shutdown, RPS software should be verified throughout the entire software development life cycle (SDLC). Recent commercial digital I&Cs (Instrumentation & Controls) use a safe-level PLC (Programmable Logic Controller) as a common hardware platform for RPS, *e.g.*, Shin Ulchin 1/2 NPPs in Korea. The RPS software is first modeled with IEC-61131-3 FBD (Function Block Diagram) [2] in the design phase. In implementation, the FBD programs are translated into C programs and then compiled into executable machine code for RPS hardware - PLC. Compiler expert companies typically provide C compilers in which functional correctness is thoroughly verified and demonstrated. Translators from FBDs to C programs are usually developed by PLC vendors. They should sufficiently demonstrate correctness and functional safety [3] of the so-called '*FBD-to-C*' translator.

Vendors such as *AREVA*[1], *invensys*[2] and *POSCO ICT*[3] have provided PLCs and software engineering tool-sets. '*SPACE*' [4] is a software engineering tool-set for *AREVA*'s PLC '*TELEPERM XS*' [5]. It stores FBD programs into a database '*INGRES*' and generates ANSI C programs for code-based testing and simulation ('*TXS SIVAT*' [6]). *ISTec GmbH*[4] has also developed a reverse engineering tool '*RETRANS*' [7] for checking the consistency between FBD programs and generated C programs. The mechanical translator in '*SPACE*' has been validated in such ways, and the software engineering tool-sets have been used successfully for more than a decade. It is worth noting that '*SPACE*' does not use a common C translator for '*RETRANS*' and executable PLC code generation. ('*TXS SIVAT*' uses two ones for different use.) PLCs of *invensys* have also been widely used. '*TriStation 1131*' [8] is its software engineering tool-set. It provides enhanced emulation-based testing and real-time simulation of FBDs, but does not include a translator into C programs.

KNICS (Korea Nuclear Instrumentation and Control System R&D Center) project [9] and *POSCO ICT* in Korea have recently developed a safety-level PLC '*POSAFE-Q*'

[1] *AREVA* (http://www.areva.com)
[2] *invensys* (http://iom.invensys.com)
[3] *PONU Tech* (http://www.ponu-tech.co.kr) for PLC segment was split from *POSCO ICT* (http://www.poscoict.co.kr)
[4] *ISTec GmbH* (http://www.istec.de)

and its software engineering tool-set '*pSET*' [10]. The tool-set provides a graphical editor for FBD and LD (Ladder Diagram) programming languages [2], and also automatically generates the ANSI C program. However, sufficient demonstration of the correctness and functional safety of the so-called '*FBD-to-C*' translator is still in progress as it is considered to be one of the most critical obstacles that must overcome in order to obtain permission for the export of the new Korean nuclear power plant [11] as a whole, *i.e.*, including control software - I&C (Instrumentation & Control).

This paper presents a technique for the development of an '*FBD-to-C*' translator, guaranteeing their fundamental behavioral equivalence without the aid of simulation and testing techniques. We propose two sets of algorithms and rules, which translate FBDs in the design phase into ANSI C programs in the implementation phase. The proposed translations use only a restricted subset of C programming language and do not need a full-scale verification, typically used in the discipline of programming [12]. We also implemented a prototype of the '*FBD-to-C*' translator and performed a case study with an example of RPS recently developed in Korea. It is not the final version of the RPS software, against which government authorities have been evaluating for years, but a preliminary one developed for the purpose of diversity and prototyping. We generated FBD programs mechanically from formal requirement specifications [13,14] as explained in [15]. The example, however, is sufficient to demonstrate that the proposed translation techniques work on all types of shutdown logics to which the RPS should give consideration.

The paper is organized as follows: Section 2 gives an introduction to a typical development process for RPS software and several techniques for validating '*FBD-to-C*' translators. We focus on *AREVA*, France and *POSCO ICT*, Korea. In section 3, we briefly introduce formal definitions of FBDs, which are pertinent to our discussion. Section 4 explains translation algorithms and rules from FBDs into a subset of ANSI C programs. In order to aid understanding, we explain the rules with an example, a basic shutdown logic; '*fixed set-point rising trip*.' Section 5 presents the result of the case study, encompassing all shutdown logics of a preliminary version of Korean APR-1400 RPS. It also demonstrates that our translation can apply to all logics of RPSs efficiently and sufficiently. Finally, Section 6 concludes the paper and provides remarks on future research extension and direction.

2. THE RPS SOFTWARE DEVELOPMENT PROCESS

RPS (Reactor Protection System) is a real-time embedded system, implemented on the hardware - PLC (Programmable Logic Controller). The RPS software is designed in FBD/LD languages and then translated into C programs which will be compiled and loaded on PLCs. Fig.1 explains a typical software development process for RPSs as well as the techniques used for validating the '*FBD-to-C*' translator.

The upper part describes a typical software development process as a waterfall model [16]. The SRS (Software Requirements Specification) is written in natural languages or formal specification languages [15,17,18]. Experts on PLC programming languages then translate the requirement specifications manually into design models programmed in FBD or LD. PLC vendors provide their own automatic translators from FBD/LD programs into ANSI C programs, while typically using COTS (Commercial Off-the-Shelf) software such as '*TMS320C55x*' of *Texas Instruments* [19] for C compilers. The COTS compilers were well verified and sufficiently certified to be used for implementing the RPS software without additional effort.

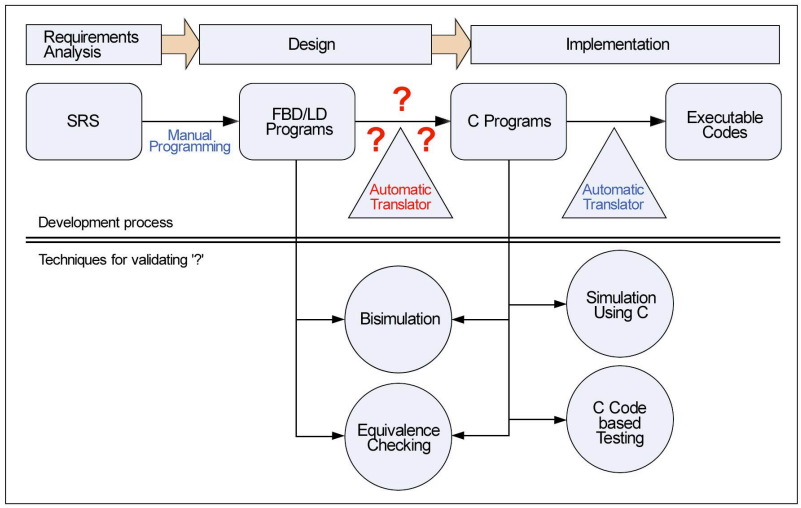

Fig. 1. A Software Development Process for RPS

The lower part of the figure shows V&V (Verification and Validation) techniques which have been used to demonstrate the correctness and functional safety of the '*Automatic Translator*.' The '*TXS SIVAT*' [6] from *AREVA*'s '*TELEPERM XS*' is an example of the C code-based simulation technique, while the '*RETRANS*' [7] is that of the bi-simulation technique. Structural testing techniques with coverage criteria [20] can also be applied into the automatically translated C programs. The KNICS project also used the tool 'IBM Rational Rhapsody' [21] to mechanically generate test cases from UML models. Equivalence checking is a verification technique recently proposed by our research team [22]. It uses a model checker *HW-CBMC* [23], which reads a Verilog program and an ANSI C program and thoroughly checks their behavioral equivalence [24]. These various techniques spanning from simulation and testing to formal verification all have been used to guarantee the correct functioning of the PLC vendor-specific '*Automatic Translator*,' *i.e*., the '*FBD-to-C*' translator. This paper proposes to use a well-verified translator from FBD to C. It can solve the fundamental consistency problem between the FBD and C programs, since the FBDtoC translator can guarantee their behavioral consistency from the viewpoint of transformation theory.

3. FUNCTION BLOCK DIAGRAM

3.1 Overview

An FBD (Function Block Diagram) consists of an arbitrary number of function blocks, '*wired*' together in a manner similar to a circuit diagram. The international standard IEC 61131-3 [2] defined 10 categories and all corresponding function blocks. Fig.2 illustrates 5 basic categories. For example, the function block ADD performs the arithmetic addition of n+1 for IN values and stores the result in the OUT variable. Others are interpreted in a similar way.

A part of preliminary FBD programs for the KNICS RPS BP (Bistable Processor) can be found in Fig.3. It was developed by domain experts from a formal requirements specification [13], and this paper uses the

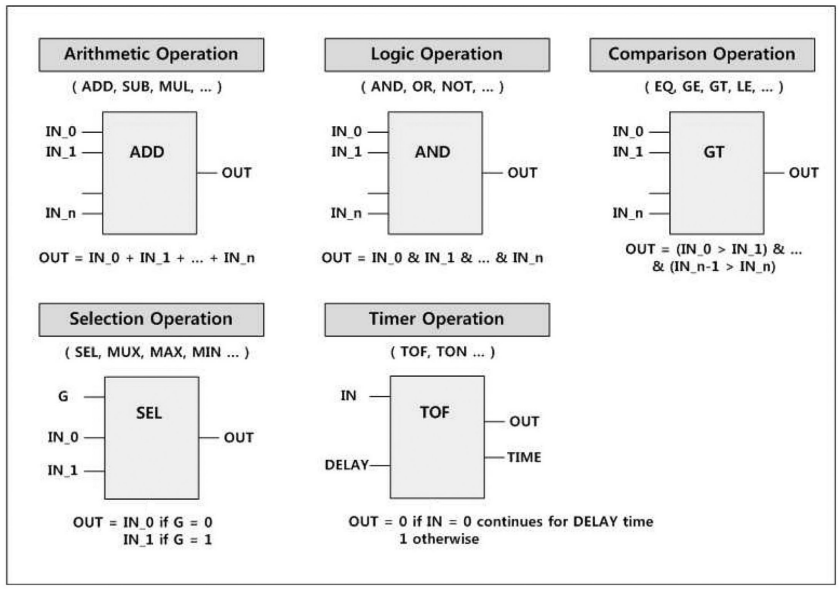

Fig. 2. Function Blocks and Categories Defined in IEC 61131-3

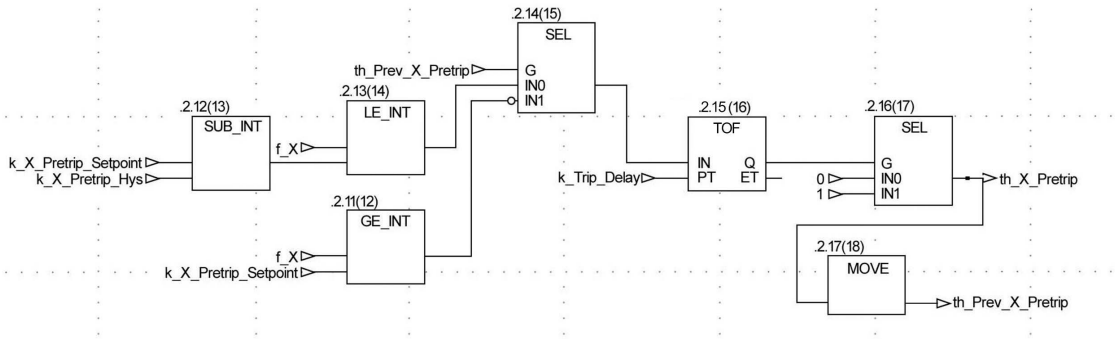

Fig. 3. An FBD for *th_X_Pretrip* Logic, Developed by Domain Experts Manually

FBD as an example throughout the paper in order to remain consistent with our former research work and aid understanding. It creates a signal '*th_X_Pretrip*' when a trip condition remains true for *k_Trip_Delay* time units as implemented with the TOF function block. It is a warning signal fired before the real one (*i.e., th_X_Trip*). The number in parenthesis above each function block denotes its execution order. Therefore, GE_INT numbered (12) is the first function block executed, while MOVE numbered (18) is the last one. The output *th_Prev_X_Pretrip* from MOVE stores the current value of *th_X_Pretrip* to use at the next execution cycle. A large number of FBDs similar to Fig.3 are assembled hierarchically and executed according to a predefined sequential execution order. The following subsection formally defines FBDs to aid in developing translation algorithms and rules from FBDs to C programs in Section 4.

3.2 Formal Definition of FBDs

This subsection formally defines FBDs in accordance with the guide of denotational/operational semantics [25] of programming languages, of which more details can be found in [26]. We define the FBD programming language as a state transition system consisting of sub-components including basic blocks in a bottom-up manner, since an FBD is a network of function blocks sequentially executed.

A ***Function Block*** is defined as a tuple composed of a name Name, input ports IP, output ports OP and its behavior description BD, as defined below. It is a basic unit for defining more complex blocks. A function block *FB* is defined [25] as a function f_{FB} from input values I_{FB} to output values O_{FB}, f_{FB}: $I_{FB} \rightarrow O_{FB}$.

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*
- *OP* : *a set of output ports, usually one*
- *BD* : *behavioral description* $\sum(p_{FB}, a_{FB})$*, where*
  - p_{FB} : *a predicate on IP*
  - a_{FB} : *assignments on OP*

Fig.4 shows definitions of ADD, SEL and TOF function blocks. We do not restrict the form of predicates and assignments on ports in *BD*. In case of the TOF timer function block, we used the TTS (Timed Transition System) [27] description, but any other formalisms such as Boolean expression, propositional logic and timed automata [28] may be used. Timer function blocks such as TOF and TON require internal storage for calculating elapsed times, and the TTS description can express it implicitly. It, however, will be defined explicitly with a local clock (internal timer) when defining a whole software system. When developing the translator from FBD to C, TOF and TON will use C library functions to get current time and do synchronized operations.

A ***Component FBD*** is a logically interconnected set of function blocks, showing an independent logical functionality. The so-called '*user-defined function block*' in the PLC industry corresponds to the component FBD. There is no strict rule for dividing FBDs into smaller elements. We should depend on our own experience, and the translation rules and algorithm can be coped with any dividing schemes. In case of the KNICS APR-1400 RPS BP, each shutdown logic (about 20 in total) can correspond to a component FBD. A component FBD is defined as a tuple composed of 4 elements: 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, $V_{Comp\_FBD\text{-}I}$ come from other component FBDs or system input variables. $V_{Comp\_FBD\text{-}O}$ denotes a set of output variables outgoing from the component FBD. The FBD depicted in Fig.3 models a fixed set-point pretrip logic for *th_X_ Pretrip* output, and the definition below regards it as a component FBD.

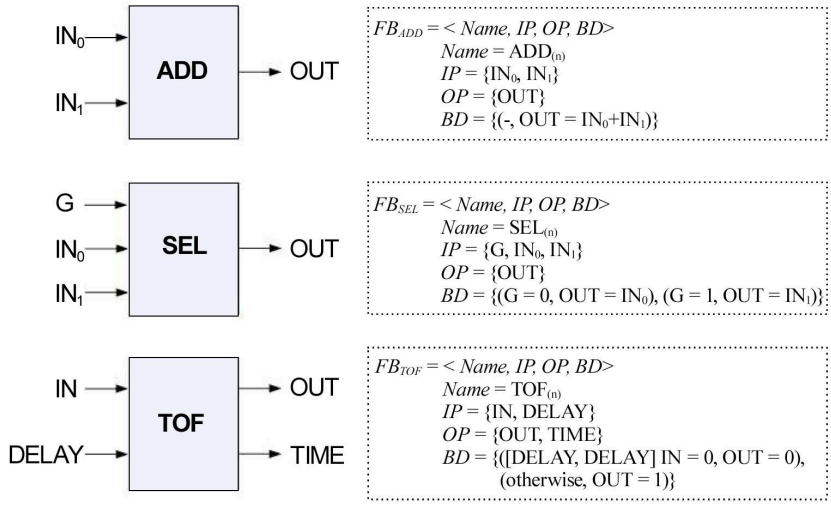

Fig. 4. Definitions of ADD, SEL, TOF Function Blocks

Definition 2 (Component FBD) *A component FBD is defined as a tuple Component_FBD = < FBs, T, I, O >, where*

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

Behavior of a component FBD is defined as a function from a set of input variables to output variables of the component FBD. Variables in $V_{Comp\_FBD\text{-}I}$ are assigned to input ports in *I*, and those in $V_{Comp\_FBD\text{-}O}$ are also assigned to output ports in *O*. Suppose that I_{Comp_FBD} is a set of input domains of input variables in $V_{Comp\_FBD\text{-}I}$ and O_{Comp_FBD} is that of $V_{Comp\_FBD\text{-}O}$, respectively, then the component FBD is defined as a function: $f_{Component\_FBD}$: $I_{Comp\_FBD} \rightarrow O_{Comp\_FBD}$.

A whole software programmed in FBD is structured with a number of component FBDs and their implicit interconnections. A ***System FBD*** defines the entire software as a tuple composed of 4 elements: component FBDs Component_FBDs, a set of transitions T between component FBDs, a set of input ports I, and a set of output ports O as follows. $V_{Sys\_FBD\text{-}I}$ denotes a set of system input variables, while $V_{Sys\_FBD\text{-}O}$ denotes a set of system output variables.

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

- *Component_FBDs* : *a set of component FBDs component_FBDs*
- *T* :
  - *a set of transitions* ($FBD_1.O_i$, $FBD_2.I_j$) *between Component FBDs* FBD_1 *and* FBD_2 *in FBDs (provided that* $FBD_1.O_i$ *is an i-th output port of* FBD_1 *and* $FBD_2.I_j$ *is a j-th input port of* FBD_2)
  - $\forall(FBD_1.O_i, FBD_2.I_j) \in T$, FBD_1 *has a sequential execution precedence on* FBD_2
- *I* : *a set of FBD's input ports FBD.I, which do not appear in T and are assigned by* $V_{Sys\_FBD\text{-}I}$
- *O* : *a set of FBD's output ports 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 $V_{Sys\_FBD\text{-}I}$ to a set of system output variables $V_{Sys\_FBD\text{-}O}$. Suppose that I_{Sys_FBD} is a set of input domains of the input variables in $V_{Sys\_FBD\text{-}I}$, and O_{Sys_FBD} is that of $V_{Sys\_FBD\text{-}O}$, then the system FBD is defined as a function: f_{System_FBD}: $I_{Sys\_FBD} \rightarrow O_{Sys\_FBD}$. A whole FBD program is now defined as a function from inputs to outputs, but we need to take into account a couple of factors in order to define the FBD system precisely.

We finally define the ***FBD Software System*** on the basis of the *System* FBD. In order to precisely define the entire behavior, we have to consider the PLC's system clock and a number of local clocks for timer function blocks, since it is real-time embedded software. An *FBD Software System* accepts inputs I_{Sys_FBD} from the external environment, performs calculations on them, and emits outputs O_{Sys_FBD} to the environment. Its behavior is the same as the system FBD. We define the behavior of an FBD software system with the function f_{System_FBD}, and there exists a transition relation *R* between system states, corresponding to $O_{Sys\_FBD} = f_{System\_FBD}(I_{Sys\_FBD})$. For each time an FBD software system emits outputs, it changes its internal system states according to its behavior defined in the system FBD. An FBD software system operates periodically with system scan cycle time *d*. The execution repeats in every time interval *d* as defined below.

Definition 4 (FBD Software System) *An FBD software system is defined as a tuple FBD_Software_System = < S, S₀, R, d >, where*

- *S* : *a set of system states,* $\sigma[\ V_{Sys\_FBD\text{-}O\text{-}Internal} \times V_{Timer}\ ]$
  - $V_{Sys\_FBD\text{-}O\text{-}Internal}$: *a set of internal output variables in* $V_{Sys\_FBD\text{-}O}$
  - V_{Timer} : *a set of timer function blocks*
- S_0 : *an initial state in S*
- *R* : *a set of transition relation* $S \times I_{Sys\_FBD} \rightarrow S' \times O_{Sys\_FBD}$
  - $O_{Sys\_FBD} = f_{System\_FBD}(I_{Sys\_FBD})$
- *d* : *a system scan cycle time*

The variable *th_Prev_X_Pretrip* in Fig.3 is an example of the $V_{Sys\_FBD\text{-}O\text{-}Internal}$, while *th_X_Pretrip* belongs to $V_{Sys\_FBD\text{-}O}$. It is not a variable in $V_{Sys\_FBD\text{-}O\text{-}Internal}$ as it is an external output outgoing to the environment. It is worth noting that timer function blocks have numerous internal states, so the state space of $[V_{Timer}]$ could extend exponentially. To avoid state explosion in formal verification, we may appropriately abstract the maximum state space of timer function blocks as introduced in [26]. In case of this paper, however, the state explosion problem does not matter, since the C programming language has more expressive power than the FBD programming language. Besides timer function blocks, there are several other IEC 61131-3 categories which have internal states to store information, *e.g.*, Bistable and Counter function blocks. They can be defined and used in a similar manner as the timer function blocks above.

4. TRANSLATIONS FROM FBD TO C

This paper proposes two ways to translate FBDs into C programs, *i.e.*, '***forward translation***' and '***backward translation***.' The translated C programs are then compiled

into executables and loaded on PLC hardware. Fig.5 overviews the translation algorithms in flowchart form. The two algorithms all start with defining C functions for all individual function blocks used in the FBD programs. Two major points make them different from each other.

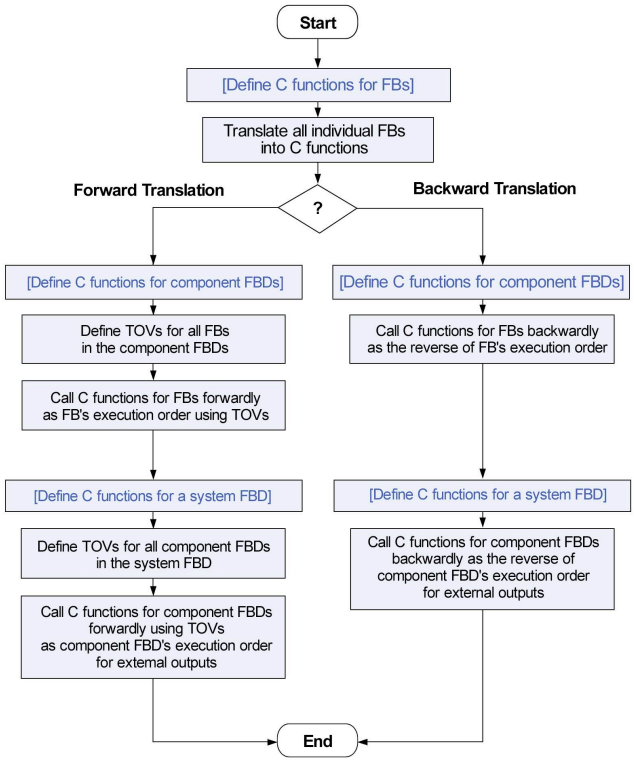

Fig. 5. An Overview of the *FBDtoC* Translation Algorithms

The one is the direction of calling C functions - forward or backward against the execution order of function blocks. The other is whether they use temporary output variables (TOVs) or not. We explain the backward translation first.

4.1 The Backward Translation

An FBD performs a data flow-based calculation according to a predefined execution order of function blocks consisting of the FBD. As the example of Fig.3, each function block has its own execution order. GE_INT numbered (12) is the function block first executed while MOVE numbered (18) is the last one. The backward translation algorithm basically works from the last function block executed to the one first executed. However, it may result in poor readability and understandability, since an FBD is composed of a number of function blocks interconnected sequentially, hierarchically and compositionally. It, however, produces more compact C programs than the forward translation.

4.1.1 Translation of Function Blocks

The rules in Fig.6 define how to translate a function block into a corresponding C function. For each function block, a matching C function is defined and called in *main* ( ) or higher level functions several times according to its sequential execution order. The [*type*] of a function *Name* corresponds to the type of output variable in *OP*. The output will be returned at the end of the function definition. The value of the output variable is assigned by the behavioral description *BD* in three ways according to their types (*i.e.*, arithmetic and logical, selection, and timer operation) as defined in rules 2 ~ 4, respectively.

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

```
[type] Name ( [type] ip0 , ··· , [type] ipn )
{
   [type] op0;
   ···   // will be filled with BD
   return op0;
}
```

2. for the *BD* of arithmetic or logical operation $\oplus$:

$op_0 = ip_0 \oplus ip_1 \oplus \cdots \oplus ip_n$

3. for the *BD* of selection operation :

```
if ( ip0 == 0 )
   op0 = ip1
else if ( ip0 == 1 )
   op0 = ip2
···
else if ( ip0 == n-1 )
   op0 = ipn
```

4. for the *BD* of timer operation :

```
int timer_started = 0;

if (ip0 == 0 && timer_started == 0)
{
   start_timer ( );
   elapsed_time ( );
   timer_started = 1;
}
else if (ip0 == 0 && timer_started == 1)
{
   if ( elapsed_time ( ip1) == 1 )
      op0 = 0;
}
else
{
   op0 = 1;
   timer_started = 0;
}
```

Fig. 6. Translation Rules for Function Blocks

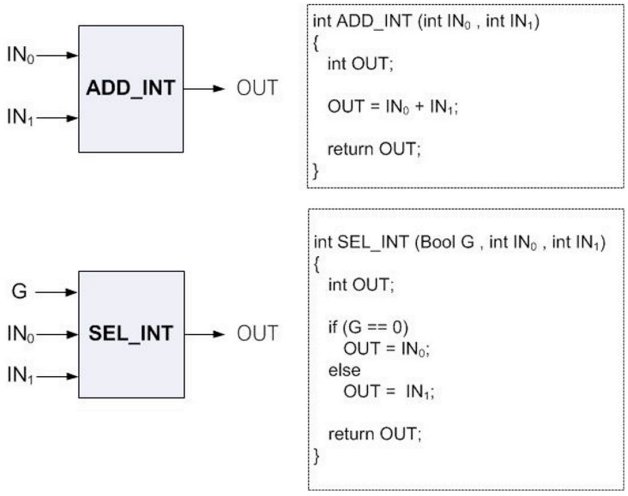

Fig. 7. An Example of Function Block Translation for ADD and SEL

5. for each *Component_FBD* = < *FBs, T, I, O* > :

[type] $op_1, \cdots, op_n$; // for internal output variables

[type] *Name* ([type] ip_0)
{
[type] op_0; // for an external output variable

⋯ // will be filled by *T*

return op_0;
}

6. for all transition relations in *T* :

op_0 = *Name_FB*$_n$ (*Name_FB*$_{n-1}$ (ip_{n-1}) , ip_n); // a recursive function

7. for all internal output variables in *O* :

op_i = *Name_FB*$_i$ (*Name_FB*$_{i-1}$ (ip_{i-1}) , ip_i); // a recursive function

Fig. 8. Translation Rules for Component FBDs

The timer operation defined in Rule 4 has more complex controls than others. Since it generates 0 value of op_0 only if ip_0 has the value of 0 for at least ip_1 time, it needs internal variables such as '*timer started*' to store the current status and elapsed time. User defined functions such as '*start timer* ( )' and '*elapsed time* ( )' should be defined concretely for industrial use. The rule above shows only the basic control flow required.

Fig.7 shows an example translating ADD and SEL function blocks. We added a type of output (such as INT for integer) to the function blocks. In practice, PLC software engineering tools provided by PLC vendors keep different function blocks for different output types, *e.g.*, ADD_INT and ADD_BOOL. The number of inputs can also differently define the same function blocks. For example, ADD_INT_2 indicates the ADD_INT for 2 inputs, while ADD_INT_4 does the same for 4 inputs. Other function blocks which are not introduced in Fig.2 can be translated into corresponding C functions in a similar manner.

4.1.2 Translation of Component FBDs

Function blocks are composed into a component FBD according to their sequential execution orders. The FBD presented in Fig.3 is a simple case of component FBD, composed of 7 function blocks. A component FBD is translated into a C function, which calls C functions of basic function blocks sequentially according to their execution orders while passing inputs and outputs. The rules in Fig.8 define the translation of component FBDs.

A component FBD has one external output and several internal outputs, as defined in Section 3. Even if internal output variables look similar to external outputs, their usage and purpose are different from external outputs. They are commonly used to store information and to be used in later execution cycles. Rule 5 translates internal output variables into global variables which are defined out of the scope of the component's definition, since they are used and called across multiple executions.

The behavior of a component FBD is defined as the sequential backward calls of all function blocks as defined in *T*. For example, the component FBD in Fig.3 consists of 7 function blocks with 6 transition relations in *T*. The last function block executed is the MOVE generating *th_Prev_X_Pretrip*, and the first one is GE_INT with two inputs, *i.e.*, *f_X* and *k_X_Pretrip_Setpoint*. The output from the last function block looks like an external output, however it is an internal output in this case and defined separately next to the '*real*' external output (in this example, *th_X_Pretrip*) as explained in rules 6 and 7. In the component FBD above, therefore, the last function block executed for the external output variable is SEL. [26] proposed a couple of criteria for internal outputs and external outputs, and we can differentiate them mechanically.

Fig.9 is a translation example for the component FBD depicted in Fig.3. We assumed that all basic function blocks consisting of the FBD were defined prior to it. For convenience, we also defined all constants with '*#define.*' The FBD's internal output variable (*th_Prev_X_Pretrip*) is defined as a global variable in line 04, and the function definition corresponding to the external output variable (*th_X_Pretrip*) is then followed in line 06. Line 09 defines the external output variable, and from line 11 to 13 the value of the external output variable is assigned through a set of function calls from the last to the first. The definition for the internal output variable is followed in line 15. The value of the external output variable is returned in line 17.

4.1.3 Translation of System FBDs

A system FBD is a set of component FBDs connected according to their sequential execution orders. Translation

```
00  #define k_X_Pretrip Setpoint 1000;
01  #define k_X_Pretrip_Hys 10;
02  #define k_Trip_Delay 5;
03
04  Bool th_Prev_X_Pretrip;
05
06  Bool Fixed_Setpoint_Rising_Trip ( int f_X )
07  {
08
09      Bool th_X_Pretrip;
10
11      th_X_Pretrip = SEL ( TOF ( SEL ( th_Prev_X_Pretrip ,
12            LE_INT ( f_X , SUB_INT ( k_X_Pretrip_Setpoint , k_X_Pretrip_Hys ) ) ,
13            ! GE_INT ( f_X , k_X_Pretrip_Setpoint ) ) , k_Trip_Delay ) , 0 , 1 ) ;
14
15      th_Prev_X_Pretrip = MOVE ( th_X_Pretrip );
16
17      return th_X_Pretrip;
18
19  }
```

Fig. 9. An Example of Translating the Component FBD in Fig.3

8. for a *System_FBD* = < *Component_FBDs, T, I, O* > :

[type] Name ([type] ip_0 , ⋯ , ip_n)
{
[type] op_0 , ⋯ , op_m // for all external output variable

⋯ // will be filled by *T*

return op_0 , ⋯ , op_m
}

9. for all transition relations in *T* and all external outputs in *O* :

op_0 = *Name_Component_FBD*$_n$ (*Name_Component_FBD*$_{n-1}$ (ip_{n-1}) , ip_n);
// a recursive function

Fig. 10. Translation Rules for System FBDs

rules for system FBDs are basically the same as those of component FBDs, but the former is composed of sequential function calls of component FBDs not individual function blocks. If the outputs of component FBDs are used as inputs of other component FBDs, as defined in *T*, they are called according to their execution orders in a reverse order. If the whole software system is structured with only one system FBD, then the C function for the system FBD corresponds to the '*main*' function of the C program. If a system FBD has many external outputs, all C functions for other component FBDs, which do not produce external outputs of the system FBD, should be executed according to their execution orders. Of course, in practice, we may have to implement the set of external outputs as a '*structure*' data type in C programming language. The translated C program for a fixed set-point shutdown logic *f_LO_SG1_LEVEL_Ptrp_Out* in the Appendix explains the above cases well. Fig.10 below summarizes the translation rules for system FBDs.

It is worth noting that the actual translation in practice may be different from the rules above and the algorithm in Fig.5. The rules above do not use temporary output variables, but just call all C functions for component FBDs backwardly. Our experiment and case study in Section 5, however, found that in-depth recursive function calls of the so-called '*pure*' backward translation makes the C program almost impossible to read and understand. We suggest using temporary variables for component FBDs and input variables for convenience. The translated C program in the Appendix shows how difficult it is to understand the '*pure*' backward calls.

4.1.4 Correctness of the Backward Translation

Correctness of the proposed translation algorithm and rules can be proven by showing that the FBD software system (Definition 4) has the same I/O behavior with the translated C program for all inputs.

Theorem 1. [Correctness of the Backward Translation] *The FBD software system always shows the same I/O behavior with the C program translated in accordance with the backward translation rules.*

Proof. The FBD program and C program have the same inputs and outputs. For all inputs, they both have the same procedures for calculating outputs, since the C program translates all the transition relations defined in *T* of component FBDs and a system FBD, backwardly. Therefore, both are equivalent and demonstrate the same I/O behavior through bi-simulation [29].

4.2 The Forward Translation

Forward translation of FBDs into C programs is quite similar with the backward translation, but uses temporary output variables (TOVs) additionally. The backward translation is intuitive to understand since it follows the execution flow backwardly from outputs to inputs. However, several depths of recursive function calls make it difficult to thoroughly understand. The forward translation uses sufficient temporary variables to understand the C programs more easily.

4.2.1 Translation of Function Blocks

Translation of function blocks is the same as the backward translation, as shown in Fig.5.

4.2.2 Translation of Component FBDs

Fig.11 defines rules for component FBDs. These are basically the same as the backward translation, but the output value is calculated and assigned forwardly as defined in Rule 5-1. For each function block, we define a TOV (*i.e.*, to_i) which stores the value only for the purpose of interconnection. Each function block is executed according to its execution order and an output value is then stored in its own TOV, as defined in Rule 6-1. The external output variable's value can be assigned with one of an appropriate TOV. Internal output variables are also defined similarly as defined in Rule 7-1.

5-1. for each *Component_FBD* = < *FBs, T, I, O* > :

[type] op_1, ⋯ , op_n; // for internal output variables

[type] *Name* ([type] ip_0)
{
[type] op_0; // for an external output variable
[type] to_0, ⋯ , to_k; // for all function blocks

⋯ // will be filled by *T*

return op_0;
}

6-1. for all transition relations in *T* and corresponding temporal output to_i :

$to_0 = Name\_FB_0 (ip_0 , \cdots);$
$to_1 = Name\_FB_1 (ip_0 , \cdots , to_0);$
...
$op_0 = Name\_FB_n (\cdots);$

7-1. for all internal output variables in *O* :

$op_i = to_i ;$

Fig. 11. Translation Rules for Component FBDs (Forward Translation)

```
00  #define k_X_Pretrip Setpoint 1000;
01  #define k_X_Pretrip_Hys 10;
02  #define k_Trip_Delay 5;
03
04  Bool th_Prev_X_Pretrip;
05
06  Bool Fixed_Setpoint_Rising_Trip ( int f_X )
07  {
08
09      Bool th_X_Pretrip;
10      int to_0, to_1, to_2, to_3, to_4, to_5, to_6;
10
11      to_0 = GE_INT ( f_X , k_X_Pretrip_Setpoint );
12      to_1 = SUB_INT ( k_X_Pretrip_Setpoint , k_X_Pretrip_Hys );
13      to_2 = LE_INT ( f_X , to_1 );
14      to_3 = SEL ( th_Prev_X_Pretrip , to_2 , ! to_0 );
15      to_4 = TOF ( to_3 , k_Trip_Delay );
16      to_5 = SEL ( to_4 , 0 , 1 );
17      to_6 = MOVE ( to_5 );
18
19      th_X_Pretrip = to_5;
20
21      th_Prev_X_Pretrip = to_6;
22
23      return th_X_Pretrip;
24
25  }
```

Fig. 12. An example of Forward Translation of the Component FBD in Fig.3

8-1. for a *System_FBD* = < *Component_FBDs, T, I, O* > :

[type] Name ([type] ip_0 , ⋯ , ip_n)
{
[type] op_0 , ⋯ , op_m // for all external output variables
[type] to_0 , ⋯ , to_k // for all definitions for component FBDs

⋯ // will be filled by *T*

return op_0 , ⋯ , op_m
}

9-1. for all transition relations in *T* and all external outputs in *O* :

$to_0 = Name\_Component\_FB_0 (ip_0 , \cdots);$
$to_1 = Name\_Component\_FB_1 (ip_0 , \cdots , to_0);$
...
$op_0 = Name\_Component\_FB_n (to_0 , ...);$
...

Fig. 13. Translation Rules for System FBDs (Forward Translation)

Fig.12 is an example of the C program translated forwardly according to the rules 5-1 ~ 7-1. It uses 7 TOVs (such as to_0 ~ to_6) as it is structured with 7 function blocks. All function blocks are executed according to predefined execution orders as lines 11 ~ 17. In case of the FBD depicted in Fig.3, the natural number in parentheses is the block's execution order. After sequentially executing all interconnected function blocks, one external output variable and possibly several internal output variables are simply assigned with values as line 19 and 21.

The forward translation of component FBDs requires more lines of codes than the backward translation (*e.g.*, 25 lines vs. 19 lines for the *Fixed_Setpoint_Rising_Trip*( ) function). However, it is more understandable and readable than the backward translation, since it uses enough auxiliary variables to aid understanding and is defined according to the execution flow of the component FBD. Section 5 compares these two approaches experimentally through translating a full-set of RPS logics. These RPS logics had been developed for a trial purpose in Korea.

4.2.3 Translation of System FBDs

A system FBD is structured with a sequential set of interconnected component FBDs. The translation rules for system FBDs are similar with those for component FBDs, but they call the component FBDs according to their execution order forwardly and store their output values in TOVs. A detailed example of the forward translation can be found in the Appendix. Fig.13 defines the rules.

4.2.4 Correctness of the Forward Translation

Correctness of the proposed translation algorithm and rules in Section 4.2 can be proven by showing that the FBD software system (Definition 4) has the same I/O behavior for all inputs with the C program translated.

Theorem 2. [Correctness of the Forward Translation] *The FBD software system always shows the same I/O behavior with the C program translated in accordance with the forward translation rules.*

Proof. The FBD program and C program have the same inputs and outputs. For all inputs, they both have the same procedures for calculating outputs, since the C program translates all the transition relations defined in *T* of component FBDs and a system FBD, forwardly. Therefore, both are equivalent and demonstrate the same I/O behavior through bi-simulation [29].

4.3 Practical Considerations on the Proposed Translations

When applying the translations to large and complex real-world systems such as RPSs in nuclear power plants, the following guidelines would be helpful.

- ***C code optimization***: The translated C program does not require code optimization since it intends to be implemented into PLC hardware not for formal verification purposes. The C programming language is more powerful and expressive than the design language FBD, and even more than the hardware description language such as Verilog and VHDL. Our former worker [26] for formal verification of FBD programs, the bit (space) optimization of translated code, was important to avoid the '*state explosion problem*' [30] which make the algorithmic verification easily infeasible. This case intends behavior-preserving code generation for implementation purposes, not formal verification. Code optimization, therefore, would be better to focus on readability and understandability of the code for manual inter-checking rather than on code efficiency such as LOC (Line number Of Code) and execution time. The C compilers and the target hardware PLC are typically fast enough to load translated C programs without any optimization.
- ***C code-based structural testing***: The C program translated by the proposed technique is not useful for typical C code-based structural testing, especially for control flow coverage-based testing [20]. As it calls sub-functions sequentially according to a predefined execution order, forwardly or backwardly, it has no complex control loops such as '*for*,' '*while*' and '*goto*.' Therefore, the typical control-flow based testing and coverage criteria (*e.g., statement, branch, condition and MC/DC*) are not effective for testing the translated C programs. A data-flow based testing and coverage criteria will be more helpful as analyzed in [31].
- ***C code execution***: The translated C program requires supplementary codes to be compiled, loaded and executed on PLC hardware. For example, an *FBD software system* (Definition 4) has a set of local timers and a global timer which synchronizes with a number of local clocks, and the translated C programs should take into account their synchronization. The internal output variables in Definition 4 are translated into global variables for component FBDs, (*e.g., th_Prev_X_Pretrip* in Fig.9 and Fig.12). The scoping of global variables generated for component FBDs should also be considered.
- ***Extending to C++***: Some important ideas of the object-oriented programming language [16] can be usefully adopted for translating basic function blocks. Development of real-world systems is apt to use various instances of a basic function block. For example, in case of ADD, we have to distinguish their variant, such as ADD with 2 integer inputs, 3 integer inputs and 2 Boolean inputs. A different C function should be defined and used for each variation. If we use an object-oriented programming language C++, variations can be defined more effectively. However, the choice of a programming language for PLCs depends not only on the efficiency and convenience of the language but also the correctness and safety level of its compiler, which has been proven.
- ***Vendor-specific FBDs***: PLC vendors such as *AREVA* and *POSCO ICT* maintain their own format and usage of FBD programming language. For example, the software engineering tool-set of *POSCO ICT* does not allow separated (*i.e.*, not interconnected FBDs), but rather all connected FBDs only. It also uses interchangeably a mixture of FBD and LD. We should be able to take into account the software program which is a mixture of FBD and LD programming languages.

5. CASE STUDY

We applied the proposed translation techniques to a set of FBD programs for APR-1400 RPS BP in Korea. The RPS was developed by KNICS [9] project consortium and several export contracts have been undergoing. We use as an example of our case study the FBDs, which were mechanically generated from a formal requirement specification [13] written in NuSCR [18]. The project used the formal requirement specification for modeling RPSs efficiently and correctly as well as achieving diversity of software requirement specifications. The formal specification and its supporting tool-set generate FBD programs mechanically [32], and several formal verification techniques (*e.g.*, model checking and equivalence checking) can also be applied to both the requirements and FBDs [15].

The mechanically generated FBDs in our case study are not the real ones used for developing the current (official) version of APR-1400 RPS, and were used when developing a prototype. They include approximately 50% more function blocks than the one developed manually by experts. However, they include all important and fundamental shutdown logics for the RPS. This case study sufficiently demonstrates that the translations proposed can work on real versions of FBDs as well as the mechanically translated ones.

Fig.14 shows the NuSCR specification while Fig. 15 depicts FBD programs mechanically translated from the

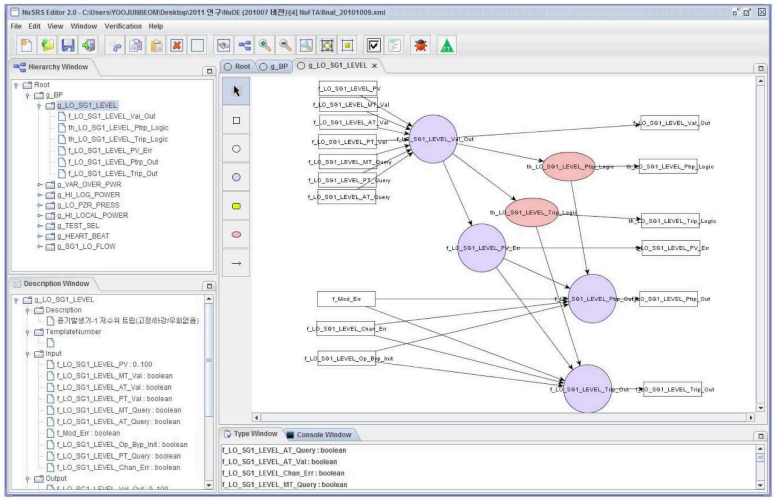

Fig. 14. NuSRS: A Tool for Supporting NuSCR Specification, Verification and FTA (ver. 2.1)

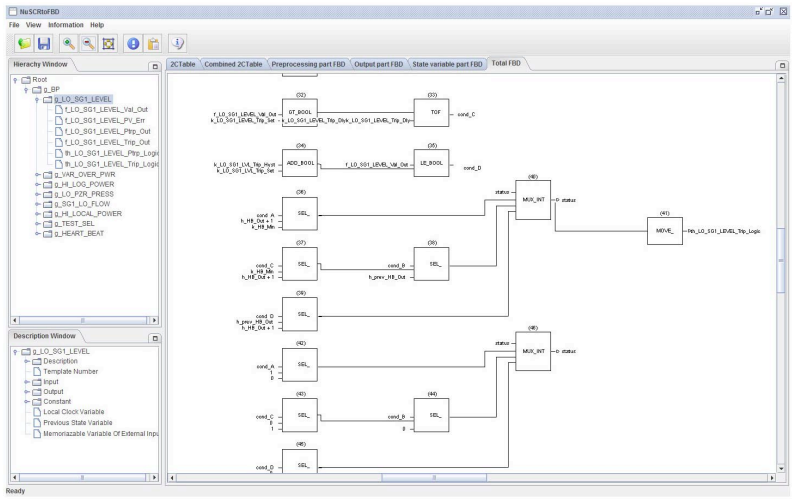

Fig. 15. NuSCRtoFBD: A Tool for Generating FBDs from NuSCR Specification (ver. 2.0)

formal requirements specification. The directory information window at the left part of the tool-sets shows 8 shutdown logics for the RPS BP, which are categorized as follows. The whole RPS BP is composed of 18 logics, but the category below encompasses all the logics.

- ***Fixed set-point logic:*** It has a fixed set-point of firing a shutdown signal. If an input value crosses the point in a rising or falling manner, the shutdown signal gets fired.
  (*e.g.*, *g_LO_SG1_LEVEL*, *g_HI_LOG_POWER*)
- ***Variable set-point logic***: It has a variable set-point of firing a shutdown signal, varying with the same (rising or falling) rate of the change of the input variable until a predefined fixed limit. If the varying rate of the input variable is more than the fixed limit, then the shutdown signal gets fired.
  (*e.g.*, *g_VAR_OVER_PWR*, *g_SG1_LO_FLOW*)
- ***Manual reset logic***: It has a fixed set-point of firing a shutdown signal, but an operator can delay the shutdown by moving the set-point to an upper point (in case of rising input flow) by pushing a reset button. The operator can push the reset button several times for specific purposes.
  (*e.g.*, *g_LO_PZR_PRESS*)

We applied the two translation techniques (the forward and backward translations) to the 6 logics of FBD programs shown in Fig.15, and compared them statically. We first developed a prototype of the FBD-to-C translator as depicted in Fig.16. Then we performed the forward and backward translations, and calculated detailed information such LOC and number of function blocks defined. It includes 4 windows; one for command information and others for three levels of translated C programs (*i.e.*, function blocks, component FBDs and a system FBD). Table 1

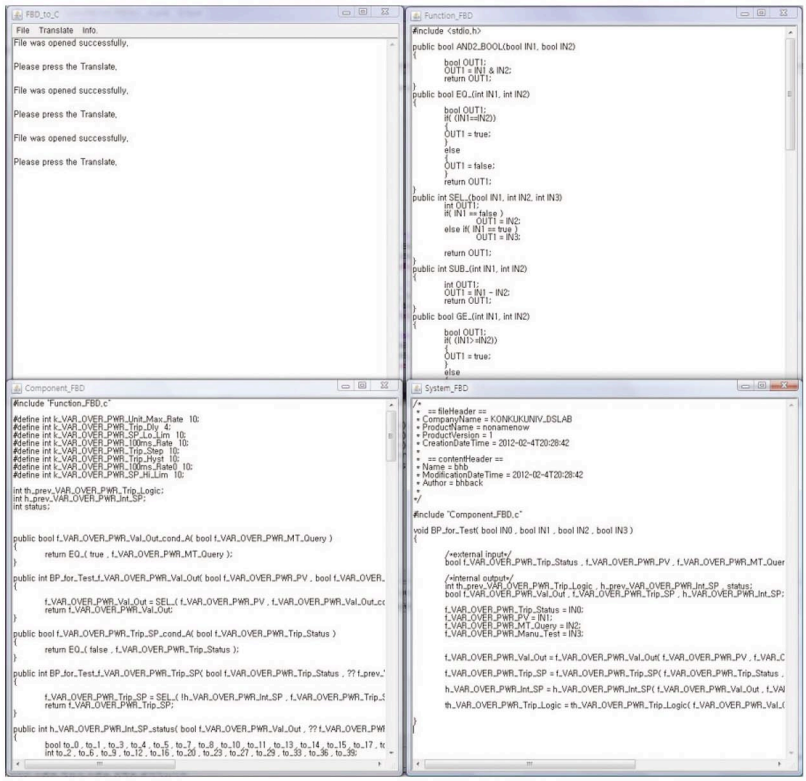

Fig. 16. A screen-dump of the *FBD-to-C* Translator (Forward Translation)

Table 1. Information of the FBDs

	Category	# of FBs	# of Comp. FBDs	# of Sys. FBDs	# of EOs	# of IOs	# of timer FBs
g_LO_SG1_LEVEL	Fixed set-point	70	31	1	2(6)	2	1
g_HI_LOG_POWER	Fixed set-point	82	27	1	2(8)	2	2
g_VAR_OVER_PWR	Variable set-point	167	45	1	2(9)	3	2
g_SG1_LO_FLOW	Variable set-point	167	45	1	2(9)	2	1
g_LO_PZR_PRESS	Manual reset	191	53	1	2(12)	4	3

presents the features of FBD programs in the RPS BP, which will be translated into C programs. As described in the table, the *fixed set-point shutdown* logic is a simple logic with relatively small function blocks, while the *manual-reset shutdown* logic is the most complicated one with about 2.5 times more function blocks than the former one. The number of external outputs for all logics in Table 1 is two, *i.e.*, trip and pre-trip signals. The FBDs, which we used as a case study, generate more than two external outputs, for testing purposes, but the actual external outputs are only the two outputs.

Table 2 compares the two translation techniques. The LOC demonstrates that the forward translation requires more codes than the backward translation, since the latter uses multi-level function calls. The Maximum depth of nested function calls points their difference out clearly. The forward translation calls function at most in a depth of three (3), while the backward translation calls function at most in a depth of six (6). The one needs more lines to unroll the nested function calls while the other needs more endurance for reading and understanding. The Number of the 18 temporary variables used can be understood in a similar way.

'*The Number of functions defined*' shows the same number for both. It reflects well on the nature of the mechanical translations. The C programs are translated from the FBD program, therefore all structure information of the higher abstraction level (*i.e.*, FBD program) are preserved in the lower level (*i.e.*, C program). For example, since the FBD for the logic *g_LO_SG1_LEVEL* is structured

Table 2. A comparison of the Forward and Backward Translations

	Translation techniques	LOC	# of functions defined	# of func. calls	Max. depth of nested func. calls	# of temporary variable used
g_LO_SG1_LEVEL	Forward	467	12+31+1	75	3	45
	Backward	541	12+31+1	78	6	6
g_HI_LOG_POWER	Forward	547	14+36+1	87	3	55
	Backward	509	14+36+1	90	7	7
g_VAR_OVER_PWR	Forward	645	13+45+1	170	3	137
	Backward	523	13+45+1	175	10	9
g_SG1_LO_FLOW	Forward	669	14+45+1	170	3	145
	Backward	543	14+45+1	175	10	9
g_LO_PZR_PRESS	Forward	653	12+53+1	195	3	158
	Backward	535	12+53+1	198	10	12

with 12 different function blocks, 31 component FBDs and 1 system FBD, the C program is also composed of 44 functions in total.

The Appendix presents two C programs translated from the FBD of the *g_LO_SG1_LEVEL* logic. We present only parts corresponding to two component FBDs and a system FBD for the pre-trip signal, due to a lack of space. Examining these two C programs convinces us of the comparison result summarized in Table 2. Comparison of these two translations from a point of dynamics, such as execution time, requires compiling and executing them on real PLC hardware. We are now planning to co-work with a PLC vendor in Korea, and also have a plan to develop a prototype translator, which is independent of specific target PLCs and compilers. Their dynamic comparison should be accompanied by a specific C compiler and target PLC hardware, and we are now planning to co-work with a PLC vendor in Korea.

6. CONCLUSION AND FUTURE WORK

Embedded software for a nuclear reactor protection system requires rigorous demonstration of safety. This paper proposes two sets of translation algorithms and rules, which translate FBD programs in the design phase into C programs in the implementation phase, while preserving their behavioral equivalence. We used an example of RPS software in a Korean nuclear power plant to demonstrate correctness and to compare the proposed translations.

We are now planning to develop a translator for a specific PLC vendor to evaluate the proposed translation techniques dynamically as well as statically. It, however, will require more elaboration on the translations which this paper proposes, since vendor-specific features should be reconsidered. In addition, we are also planning to embed a prototype of the translator, which does not take into account vendor-specific factors, into our software development framework for RPSs, *NuDE* (*Nuclear Development Environment*) [15].

ACKNOWLEDGEMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0002566). It was also supported by the MKE (The Ministry of Knowledge Economy), Korea, under the Development of Performance Improvement Technology for Engineering Tool of Safety PLC (Programmable Logic Controller) program supervised by the KETEP (Korea Institute of Energy Technology Evaluation And Planning)" (KETEP-2010-T1001-01038) and a grant from the Korea Ministry of Strategy, under the development of the integrated framework of I&C conformity assessment, sustainable monitoring, and emergency response for nuclear facilities.

APPENDIX

A translation example: C programs for fixed set-point shutdown logic

A. The Forward Translation

```
/*component FBDs*/
  ...

/*f_LO_SG1_LEVEL_PV_Err*/
BOOL PV_Err_CondA(unsigned int f_LO_SG1_
LEVEL_Val_Out){
  return LT_BOOL(f_LO_SG1_LEVEL_Val_Out, k_LO_SG1
  _LEVEL_PV_Max);
}

BOOL PV_Err_CondB(unsigned int f_LO_SG1_ LEVEL
_Val_Out){
```

```
  return GT_BOOL(f_LO_SG1_LEVEL_Val_Out, k_LO
  _SG1_LEVEL_PV_Min);
}

BOOL f_LO_SG1_LEVEL_PV_Err(unsigned int f_ LO_
SG1_LEVEL_Val_Out){
  BOOL to_0;

  to_0 = AND2_BOOL(!PV_Err_CondB(f_LO_SG1_
  LEVEL_Val_Out), !PV_Err_CondA(f_LO_SG1_ LEVEL
  _Val_Out));

  return SEL_(to_0, false, true);
}

/*f_LO_SG1_LEVEL_Ptrp_Out*/
BOOL Ptrp_Out_CondA(unsigned int th_LO_SG1_LEVEL
_Ptrp_Logic){
  return EQ_BOOL(true, th_LO_SG1_LEVEL_Ptrp _Logic);
}

BOOL Ptrp_Out_CondB(BOOL f_LO_SG1_ LEVEL_Op
_Byp_Init){
  return EQ_BOOL(false, f_LO_SG1_LEVEL_Op_ Byp
  _Init);
}

BOOL Ptrp_Out_CondC(BOOL f_Mod_Err){
  return EQ_BOOL(true, f_Mod_Err);
}

BOOL Ptrp_Out_CondD(BOOL f_LO_SG1_LEVEL_ Chan
_Err){
  return EQ_BOOL(true, f_LO_SG1_LEVEL_Chan _Err);
}

BOOL Ptrp_Out_CondE(BOOL f_LO_SG1_LEVEL_ PV
_Err){
  return EQ_BOOL(true, f_LO_SG1_LEVEL_PV_ Err);
}

BOOL f_LO_SG1_LEVEL_Ptrp_Out(unsigned int th_LO_
SG1_LEVEL_Ptrp_Logic, BOOL f_Mod_Err, BOOL f_LO
_SG1_LEVEL_Op_Byp_Init, BOOL f_LO_ SG1_LEVEL
_Chan_Err, BOOL f_LO_SG1 _LEVEL_ PV_Err){
  BOOL to_0, to_1, to_2, to_3, to_4;

  to_0 = AND2_BOOL(Ptrp_Out_CondB(f_LO_ SG1_
       LEVEL_Op_Byp_Init), Ptrp_Out_CondA (th_ LO
       _SG1_LEVEL_Ptrp_Logic));
  to_1 = OR2_BOOL(Ptrp_Out_CondE(f_LO_SG1_
       LEVEL _PV_Err), Ptrp_Out_CondD(f_LO_
       SG1_LEVEL_Chan_Err));
  to_2 = OR2_BOOL(to_1, Ptrp_Out_CondC(f_Mod _Err));
  to_3 = AND2_BOOL(to_2, to_0);
  to_4 = SEL_(to_3, false, true);

  return to_4;
}
  ...

typedef struct{
  BOOL trip;
  BOOL ptrp;
}Trip_signal;
```

```
/*system FBD*/
Trip_signal get_signal(unsigned int IN0, BOOL IN1, BOOL
IN2, BOOL IN3, BOOL IN4, BOOL IN5, BOOL IN6, BOOL
IN7, BOOL IN8, BOOL IN9, unsigned int IN10, unsigned
int IN11, unsigned int IN12, unsigned int IN13){
  Trip_signal *signal;

  /*external input*/
  unsigned int f_LO_SG1_LEVEL_PV;
  BOOL f_LO_SG1_LEVEL_MT_Query, f_LO_ SG1_
  LEVEL_AT_Query, f_LO_SG1_LEVEL_PT _Query;
  BOOL f_LO_SG1_LEVEL_PT_Val, f_LO_SG1_L
  EVEL_AT_Val, f_LO_SG1_LEVEL_MT_Val;
  BOOL f_LO_SG1_LEVEL_Op_Byp_Init, f_Mod_ Err,
  f_LO_SG1_LEVEL_Chan_Err;

/*external output variables for testing purposes*/
  unsigned int Val_Out;
  unsigned int Ptrp_Logic, Trip_Logic;
  BOOL PV_Err;

/*assignment for external input values*/
  f_LO_SG1_LEVEL_PV = IN0;
  f_LO_SG1_LEVEL_MT_Query = IN1;
  f_LO_SG1_LEVEL_AT_Query = IN2;
  f_LO_SG1_LEVEL_PT_Query = IN3;
  f_LO_SG1_LEVEL_PT_Val = IN4;
  f_LO_SG1_LEVEL_AT_Val = IN5;
  f_LO_SG1_LEVEL_MT_Val = IN6;
  f_LO_SG1_LEVEL_Op_Byp_Init = IN7;
  f_Mod_Err = IN8;
  f_LO_SG1_LEVEL_Chan_Err = IN9;

/*executions of component FBDs*/
  Val_Out = f_LO_SG1_LEVEL_Val_Out(f_LO_SG1
            _LEVEL_MT_Query, f_LO_SG1 _LEVEL
            _AT_Query,
            f_LO_SG1_LEVEL_PT_Query, f_LO_SG1_
            LEVEL_PV, f_LO_SG1 _LEVEL_PT_Val,
            f_LO_SG1_LEVEL _AT_Val,
            f_LO_SG1_LEVEL_MT_Val);

  Ptrp_Logic = th_LO_SG1_LEVEL_Ptrp_Logic (Val_Out);
               Set_Ptrp_Logic_Status(Val_Out);

  PV_Err = f_LO_SG1_LEVEL_PV_Err(Val_Out);

  signal->trip = f_LO_SG1_LEVEL_Trip_Out (Trip_Logic,
                 f_LO_SG1_LEVEL _Op_Byp_Init, f_Mod
                 _Err,
                 f_LO_SG1_LEVEL_Chan_Err, PV _Err);

  signal->ptrp = f_LO_SG1_LEVEL_Ptrp_Out (Ptrp_ Logic,
                 f_LO_SG1_LEVEL_Op_ Byp_Init, f_Mod
                 _Err,
                 f_LO_SG1_LEVEL_Chan_Err, PV_ Err);

  return signal;
}
```

B. The Backward Translation

```
/*component FBDs*/
```

```
  ...

/*f_LO_SG1_LEVEL_PV_Err*/
BOOL PV_Err_CondA(unsigned int f_LO_SG1_LEVEL
_Val_Out){
  return LT_BOOL(f_LO_SG1_LEVEL_Val_Out, k_LO
  _SG1_LEVEL_PV_Max);
}

BOOL PV_Err_CondB(unsigned int f_LO_SG1_LEVEL
_Val_Out){
  return GT_BOOL(f_LO_SG1_LEVEL_Val_Out, k_LO
  _SG1_LEVEL_PV_Min);
}

BOOL f_LO_SG1_LEVEL_PV_Err(unsigned int f_LO
_SG1_LEVEL_Val_Out){
  return SEL_(AND2_BOOL(!PV_Err_CondB(f_LO_
  SG1_LEVEL_Val_Out), !PV_Err_CondA(f_LO_ SG1
  _LEVEL_Val_Out)), false, true);
}

/*f_LO_SG1_LEVEL_Ptrp_Out*/
BOOL Ptrp_Out_CondA(unsigned int th_LO_SG1_ LEVEL
_Ptrp_Logic){
  return EQ_BOOL(true, th_LO_SG1_LEVEL_Ptrp _Logic);
}

BOOL Ptrp_Out_CondB(BOOL f_LO_SG1_LEVEL_Op
_Byp_Init){
  return EQ_BOOL(false, f_LO_SG1_LEVEL_Op_ Byp
  _Init);
}

BOOL Ptrp_Out_CondC(BOOL f_Mod_Err){
  return EQ_BOOL(true, f_Mod_Err);
}

BOOL Ptrp_Out_CondD(BOOL f_LO_SG1_LEVEL_
Chan_Err){
  return EQ_BOOL(true, f_LO_SG1_LEVEL_Chan _Err);
}

BOOL Ptrp_Out_CondE(BOOL f_LO_SG1_LEVEL_PV
_Err){
  return EQ_BOOL(true, f_LO_SG1_LEVEL_PV_ Err);
}

BOOL f_LO_SG1_LEVEL_Ptrp_Out(unsigned int th_LO
_SG1_LEVEL_Ptrp_Logic, BOOL f_Mod_ Err, BOOL f_
LO_SG1_LEVEL_Op_Byp_Init, BOOL f_LO_SG1_LEVEL
_Chan_Err, BOOL f_LO_SG1_LEVEL _PV_Err){
  return SEL_(AND2_BOOL(OR2_BOOL( OR2_ BOOL
  (Ptrp_Out_CondE(f_LO_SG1_LEVEL_PV_Err),
    Ptrp_Out_CondD(f_LO_SG1_LEVEL_Chan_Err)),
    Ptrp_Out_CondC(f_Mod_Err)),
    AND2_BOOL(Ptrp_Out_CondB(f_LO_SG1_LEVEL_
    Op_Byp_Init),
    Ptrp_Out_CondA(th_LO_SG1_LEVEL_Ptrp_Logic))),
    false, true);
  ...

typedef struct{
  BOOL trip;
  BOOL ptrp;
}Trip_signal;

/*system FBD*/
Trip_signal get_signal(unsigned int IN0, BOOL IN1,
BOOL IN2, BOOL IN3, BOOL IN4, BOOL IN5, BOOL
IN6, BOOL IN7, BOOL IN8,BOOL IN9){
  Trip_signal *signal;

/*external input*/
  unsigned int f_LO_SG1_LEVEL_PV;
  BOOL f_LO_SG1_LEVEL_MT_Query, f_LO_S G1_
  LEVEL_AT_Query, f_LO_SG1_LEVEL_PT_Query;
  BOOL f_LO_SG1_LEVEL_PT_Val, f_LO_SG1_LEVEL
  _AT_Val, f_LO_SG1_LEVEL_MT_Val;
  BOOL f_LO_SG1_LEVEL_Op_Byp_Init, f_Mod_ Err,
  f_LO_SG1_LEVEL_Chan_Err;

/*assignments for external input values*/
  f_LO_SG1_LEVEL_PV = IN0;
  f_LO_SG1_LEVEL_MT_Query = IN1;
  f_LO_SG1_LEVEL_AT_Query = IN2;
  f_LO_SG1_LEVEL_PT_Query = IN3;
  f_LO_SG1_LEVEL_PT_Val = IN4;
  f_LO_SG1_LEVEL_AT_Val = IN5;
  f_LO_SG1_LEVEL_MT_Val = IN6;
  f_LO_SG1_LEVEL_Op_Byp_Init = IN7;
  f_Mod_Err = IN8;
  f_LO_SG1_LEVEL_Chan_Err = IN9;

/*executions of component FBDs*/
  Ptrp_Logic = th_LO_SG1_LEVEL_Ptrp0 _Logic (Val
               _Out);
  Set_Ptrp_Logic_Status(Val_Out);
  PV_Err = f_LO_SG1_LEVEL_PV_Err(Val_Out);

/*assignments for external output variables*/
  signal->trip = f_LO_SG1_LEVEL_Trip_Out( ... );

  signal->ptrp = f_LO_SG1_LEVEL_Ptrp_Out( th_
  LO_SG1_LEVEL_Ptrp_Logic(
    f_LO_SG1_LEVEL_Val_Out(
      f_LO_SG1_LEVEL_MT_Query,
      f_LO_SG1_LEVEL_AT_Query,
      f_LO_SG1_LEVEL_PT_Query,
      f_LO_SG1_LEVEL_PV,
      f_LO_SG1_LEVEL_PT_Val,
      f_LO_SG1_LEVEL_AT_Val,
      f_LO_SG1_LEVEL_MT_Val)),
    f_Mod_Err,f_LO_SG1_LEVEL_Chan_Err, f_LO
    _SG1_LEVEL_PV_Err(
    f_LO_SG1_LEVEL_Val_Out(
      f_LO_SG1_LEVEL_MT_Query,
      f_LO_SG1_LEVEL_AT_Query,
      f_LO_SG1_LEVEL_PT_Query,
      f_LO_SG1_LEVEL_PV,
      f_LO_SG1_LEVEL_PT_Val,
      f_LO_SG1_LEVEL_AT_Val,
      f_LO_SG1_LEVEL_MT_Val)));

  return signal;
}
```

REFERENCES

[1] N. G. Leveson, SAFEWARE, System safety and Computers, Addison Wesley, (1995).

[2] International Electrotechnical Commission, International standard for programmable controllers: Programming languages, part 3 (1993).

[3] IEC, IEC 61508, Functional safety of electrical, electronic and programmable electronic (E/E/PE) safety-related systems, (2000).

[4] SIEMENS, Space, engineering system of teleperm xs plc, Tech. Rep. KWU NLL1-1026-76-V1.0/11.96, Germany (1996).

[5] SIEMENS, Teleperm xs, brief description, Tech. Rep. KWU NLL1-1004-76-V2.2/04.98, Germany (1998).

[6] S. Richter, J. Wittig, "Verification and validation process for safety I&C systems", *Nuclear Plant Journal*, vol. 21 (3), pp.36–40 (2003)

[7] ISTec, RETRANS, reverse engineering tool for fbd programming of teleperm xs plc, Tech. rep., Germany (1997).

[8] invensys, Safety software suite, TriStation 1131 (TS1131), http://iom.invensys.com/.

[9] KNICS, Korea nuclear instrumentation and control system R&D center, http://www.knics.re.kr/english/eindex.html.

[10] S. Cho, K. Koo, B. You, T.-W. Kim, T. Shim, J. Lee, "Development of the loader software for PLC programming", *Proceedings Conference of the Institute of Electronics Engineers of Korea*, Vol. 30 (1), pp. 959–960, (2007).

[11] WIKIPEDIA, Nuclear power in south korea, http://en.wikipedia.org/wiki/Nuclear power in South Korea.

[12] T. Hoare, "The verifying compiler: A grand challenge for computing research", *Journal of the ACM*, vol. 50 (1), pp. 63–69 (2003).

[13] Korea Atomic Energy Rearch Institute, SRS for Reactor Protection System, KNICS-RPS-SRS101 Rev.00 (2003).

[14] KAERI(Korea Atomic Energy Rearch Institute), Fromal SRS for Reactor Protection System, KNICS-RPS-SVR131 -01 Rev.00 (2005).

[15] J. Yoo, E. Jee, S. S. Cha, "Formal Modeling and Verification of Safety-Critical Software", *IEEE Software*, vol. 26 (3), pp. 42–49 (2009).

[16] I. Sommerville, "SOFTWARE ENGINEERING", 9th Edition, Addison Wesley, (2010).

[17] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw, "Automated consistency checking of requirements specifications", *IEEE Transactions on Software Engineering*, vol. 5 (3), pp. 231 –261 (1996).

[18] 18 J. Yoo, T. Kim, S. Cha, J.-S. Lee, H. S. Son, "A Formal Software Requirements Specification Method for Digital Nuclear Plants Protection Systems", *Journal of Systems and Software*, vol. 74 (1), pp. 73–83 (2005).

[19] TEXAS INSTRUMENTS, TMS320C55x optimizing c/c++ compiler users guide, Tech. Rep. SPRU281F, TEXAS INSTRUMENTS (2003).

[20] 20 M. Pezze, M. Young, "Software Testing and Analysis", WILEY (2008).

[21] IBM Rational, Rational rhapsody, http://www-01.ibm.com /software/awdtools/rhapsody/.

[22] D.-A. Lee, J. Yoo, J.-S. Lee, "Equivalence checking between function block diagrams and c programs using HW-CBMC", *The 30th International Conference on Computer Safety, Reliability and Security* (SAFECOMP 2011), pp. 397-408 (2011).

[23] E. M. Clarke, D. Kroening, "Hardware verification using ANSI-C programs as a reference", *Proceedings of the 2003 Asia and South Pacificc Design Automation Conference*, pp. 308–311 (2003).

[24] S.-Y. Huang, K.-T. Cheng, "Fromal Equivalence Checking and Debugging", Kliwer Academic Publishers (1998).

[25] R. Tennent, "The denotational semantics of programming languages", *Communicatin of the ACM*, vol. 19 (8), pp. 437–453 (1976).

[26] J. Yoo, S. Cha, E. Jee, "Verificatin of PLC Programs written in FBD with VIS", *Nuclear Engineering and Technology*, vol. 41 (1) pp. 79–90 (2009).

[27] T. Henzinger, Z. Manna, A. Pnueli, "Timed transition systems", *REX Workshop*, pp. 226–251 (1991).

[28] R. Alur, D. L. Dill, "A theory of timed automata", *Theoretical Computer Science* vol. 126 (2), pp. 183-235 (1994)

[29] J. Davoren, "Topologies, continuity and bisimulations", *Theoretical Informatics and Applications*, vol. 33, pp. 357 –381 (1999).

[30] E. M. Clarke, O. Grumberg, D. A. Peled, "Model Checking", *MIT Press*, (1999).

[31] E. Jee, J. Yoo, S. Cha, D. Bae, "A data flow-based structural testing technique for fbd programs", *Information and Software Technology*, vol. 51 (7), pp. 1131–1139 (2009).

[32] J. Yoo, S. Cha, C. H. Kim, D. Y. Song, "Synthesis of FBD based PLC Design from NuSCR Formal Specification", *Reliability Engineering and System Safety*, vol. 87 (2), pp. 287–294 (2005).