2014 21st Asia-Pacific Software Engineering Conference

Guidelines for the Use of Function Block Diagram in Reactor Protection Systems

Dong-Ah Lee, Junbeom Yoo
Division of Computer Science and Engineering
College of Information and Communication, Konkuk University
Seoul, Republic of Korea
Email: {ldalove, jbyoo}@konkuk.ac.kr

Jang-Soo Lee
Man-Machine Interface System team
Korea Atomic Energy Research Institute
Daejeon, Republic of Korea
Email: jslee@kaeri.re.kr

***Abstract*—Making software dependable is one of most important aspects in safety-critical system such as a nuclear power plant. Dependable programming techniques to get rid of undependable properties, such as ambiguity, wrong uses of elements, discordance, etc., help engineers reduce the faults in programs. This paper proposes the practice guidelines for function block diagram (FBD) which is a programming language for programmable logic controllers (PLC) widely used in industry. The guidelines show that what cases cause undependable properties and how the properties should be eliminated to be dependable ones in FBD programs. The paper introduces the application of guidelines to the trip (shutdown) logic of bistable processor (BP) in reactor protection system (RPS) developed in the Korea Nuclear Instrumentation and Control System R&D Center (KNICS). The application describes that the guidelines eliminates undependable properties in the trip logic.**

I. Introduction

Dependability of software in critical systems such as nuclear power plants and insulin pumps is one of the most important properties, because a failure of the systems may result in injuries to people, damages to the environment, or extensive economic losses [1]. Furthermore, it requires high dependability to get permissions for operation and export from government authorities. Such systems often use Function Block Diagram (FBD) to develop its embedded software. For example, POSAFE-Q Software Engineering Tool (pSET) [2] which is developed by Korea Nuclear Instrumentation and Control Systems R&D Center (KNICS) [3] used FBD to develop software of reactor protection system (RPS). The systems cannot be approved for operation by the regulation agency (e.g., KINS [4] in Korea), if the software designed using FBD satisfies high dependability as high as the regulation agency demands.

FBD is one of the widely used Programmable Logic Controllers (PLC) programming languages defined in the IEC 61131-3 standard [5]. As FBD is a graphical language, it usually requires translation into other languages such as C, Verilog [6], or specific machine code [7]–[9] for implementation, simulation, or verification. Although the standard defines usage of elements, undefined or unexpected uses show up, because development environment is much dependent upon vendors. Software development tools, developed by domain specific vendors, include their own rules which the standard does not explicitly specify about the usages. These rules may give engineers ambiguity or misunderstanding. FBD programs, which is developed using such tools, are functionally correct; however, it is possible that they operate incorrectly on other environment. Furthermore, if there is a change of environment in a domain, such as version-up of the tools, then the same program may not operate equally.

pSET uses FBD to design software for POSAFE-Q Programmable Logic Controllers (PLCs). The designed software is implemented using C language to load them onto the PLCs. We found several undependable cases which analysts cannot catch up the operation directly. Implementation programs from the design using the FBD language which includes the undependable cases operate correctly in the target system. It, however, cannot be evaluated by analysts accurately; and other environments, such as a next version of pSET or other engineering tools, may operate the software incorrectly. These potential incorrectness is able to lead systems to catastrophic accidents especially in safety-critical ones.

This paper introduces practical guidelines for FBD programming to eliminate the undependable cases which is the undefined or unexpected use of the FBD language. We show 5 undependable cases and present how the cases should be modified to be dependable. The undependable cases mean that the use of the FBD language have ambiguity, wrong uses of elements, discordance, etc. It also presents how the practical guidelines eliminate the undependable properties in the trip (shutdown) logics of bistable processor (BP), which is a part of a preliminary version of Korean APR-1400 RPS developed in the Korea Nuclear Instrumentation and Control System R&D Center (KNICS). The case study shows enhances dependability of the logic through elimination of the undependable cases, which have potential incorrectness.

The remainder of the paper is organized as follows: Section 2 explains FBD, pSET, and other dependable programming researches briefly. Section 3 introduces the 5 undependable cases for FBD programming and presents practical guidelines one by one. Section 4 shows the case study about application of the guidelines to the target software and we finally conclude the paper at Section 5.

II. Related Work

A. Function Block Diagram

FBD consists of an arbitrary number of functions and function blocks, which is ‘wired’ together in a manner similar

1530-1362/14 $31.00 © 2014 IEEE
DOI 10.1109/APSEC.2014.29

to a circuit diagram. The international standard IEC 61131-3 defines 10 categories and we present six out of the ten in <Fig. 1>. For example, the function block `ADD` performs arithmetic addition of all input values (`IN1`-`INn`) and makes a result of the function onto the `OUT` variable. Others are interpreted in a similar way.

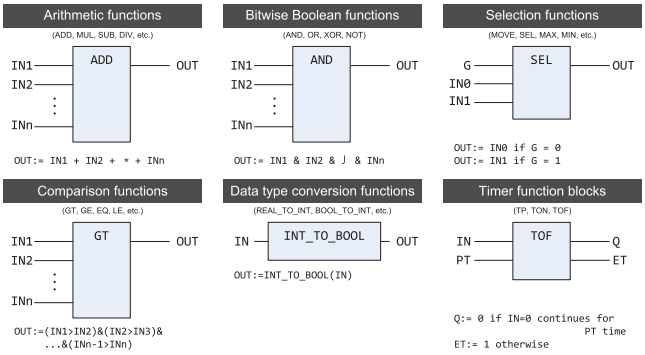

Fig. 1. Examples of Functions and Function Blocks

B. pSET

POSAFE-Q Software Engineering Tool (pSET) [2] is a loader software to develop a program of POSAFE-Q PLC and is developed as a part of the KNICS project. It satisfies IEC 61131-3 standards and support GUI environment, C language programming, monitoring/debugging functions, and simulations. pSET was developed considering NUREG/CR6463 Guideline [10] because a target domain is nuclear power plant system. <Fig. 2> shows a screen dump of the pSET.

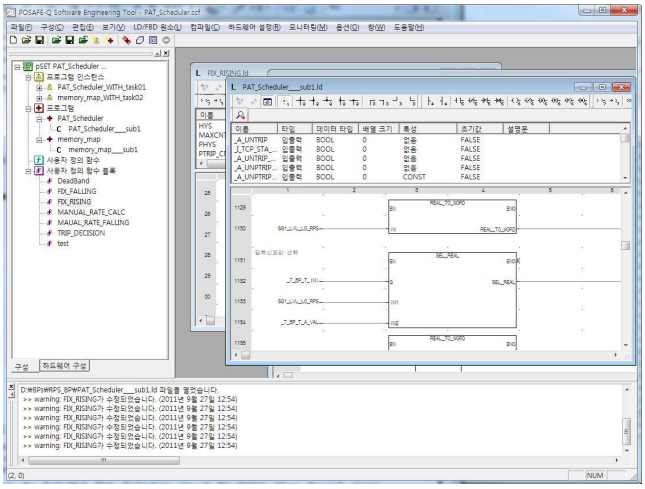

Fig. 2. A screen dump of pSET

pSET uses FBD, Ladder Diagram (LD), Sequential Function Chart (SFC) and C Code (CC) to develop PLC programs[1]. An automatic translator of pSET translates FBD programs into ANSI-C language programs to compile them for machine codes of PLC. The translation [7] has an important advantage which users are able to easily verify the programs with common verification or testing tools.

C. Dependable Programming

Most faults and failures in software are results of human errors mainly. There are many studies and regulations to reduce the human errors. [1] introduces programming structures which may cause potential errors: floating point numbers, pointers, dynamic memory allocation, and so on. Some of regulations for safety-critical system prohibit these kinds of structures. There exists programming languages, Java, which excludes structures that can easily cause errors such as goto statement or dynamic memory allocation. Java, however, still have some of the structures in it.

Regulations or standards for safety-critical systems play important role in industry. Many of them have the regulations or standards for designing or coding. For example, IEC 61508-3 [11] has design and coding standards which can be applied many different fields. Many domains of safety-critical systems also have them—DO-178B [12] for airborne systems, IEC 62304 [13] for medical device software, ISO 26262 [14] and MISRA C/C++ [15], [16] for automotive industry, IEC 60880 [17] for nuclear power plants, and so on. There also exist tools, such as LDRA [18] or SCADE suite [19], which can verify whether a program satisfies specific regulations or standards.

Many researches and regulations have contributed to dependable programming; they focus on text and control flow based languages. The big part of former studies is able to share the idea; however, they do not cover all undependable properties in the FBD language because it is a graphical and data flow based language. The paper covers undependable cases which appear in FBD programs specifically.

III. Practical Guidelines for FBD Program

This section introduces 5 undependable cases in FBD programs, and we introduce how developers prevent the cases retaining the same behavior as previous ones.

A. Guideline 1: Execution control except EN and ENO signals

FBD is a block diagram which evaluates outputs with inputs. It performs the evaluation based on data flow from inputs to outputs. Inputs and outputs are connected with wires, which may include blocks between the inputs and outputs. The wires connect two elements of FBD:

- an input to an output
- an input to an input port of a block
- an output port of a block to an input port of a block
- an output port of a block to an output

A block may have an additional ports—Boolean "`EN`" (Enable) input and "`ENO`" (Enable Out) output. IEC 61131-3 standard defines rules about the execution of the operations, when the `EN` and `ENO` ports are used. Rules consist of three situations:

- when `EN` is set to *FALSE* (0)

[1]IEC 61131-3 standard includes only FBD, LD and SFC. pSET also supports CC to apply requirements of the nuclear fields

- when EN is set to *TRUE* (1)
- when one of errors defined in error conditions occurs during the execution

The ENO is set to *FALSE* in the first and last situations. On the other hand, the ENO is set to *TRUE* when it is the second situation. They explain which value the ENO have depending on the situations; however, it is not clear that what values the output ports of the block have in those cases. Therefore, a development tool or a system for FBD decide its behavior with their regulations. For example, all outputs are set to *FALSE* or their previous value.

FBD program can use the EN/ENO ports not only to enable and disable blocks, but also to control function flow directly. <Fig. 3> shows an undependable case using EN/ENO ports as a control signal. C_TRUE in <Fig. 3> and later ones means a constant variable which is set to *TRUE* (1). The program probably intended to evaluate E as below:

- if $(A\&B) = FALSE$ then $E := C + D$
- if $(A\&B) = TRUE$ then $E := C - D$

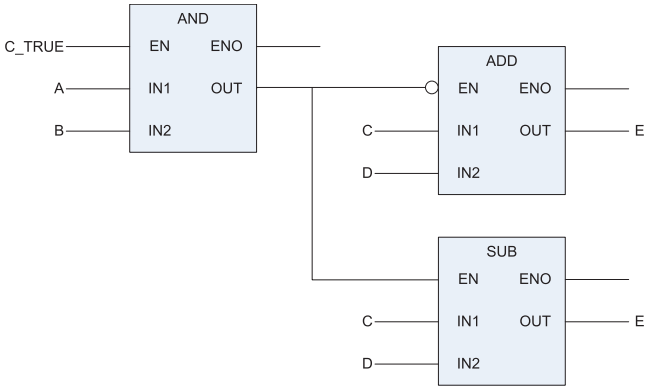

Fig. 3. Undependable case A: The unsafe case using EN/ENO as control signals

The output of AND function block controls two different function blocks, ADD and SUB. The case A shows control flow based programming which is unsuitable for FBD programming, which is based on data flow. Furthermore, analysts cannot evaluate the output E clearly, because the analyst does not sure to what value E is set when the EN port of one of two function blocks, ADD and SUB, is reset to *FALSE* (0).

To eliminate the undependable case about usage of EN and ENO as control signals, we propose guideline 1 described in <Fig. 4>. The SEL performs binary selection function which evaluate E as below

- if $G = FALSE$ then $OUT := IN1$ which $IN1 = A + B$
- if $G = TRUE$ then $OUT := IN2$ which $IN2 = A - B$

There are not any elements to confuse developers or analysts, although more function blocks and wires are in the program than <Fig. 3>. The EN/ENO ports are not mandatory to be in a block according to the standard. Therefore, we

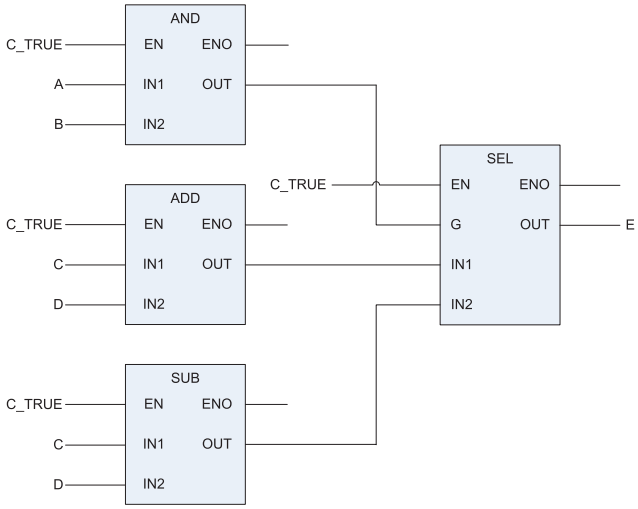

Fig. 4. Elimination of undependable case A using a SEL function

strongly recommend that FBD programs does not use EN/ENO ports to make program more dependable. We assume that all EN and ENO signals are set to *TRUE* and do not indicate them in the remained paper.

B. Guideline 2: Usage of Output Variables

FBD programs evaluate its output based on cycles which means the program receives all inputs at once and the programs evaluate all outputs at once. Next evaluation will be performed at the next cycle with next value of all inputs. For example, the output E in <Fig. 4> is set to one of values, which are E:=C+D or E:=C−D by input A and B, once in a cycle, and both input, C and D, of both blocks, ADD and SUB, are always same at a cycle. The E has only one value in a cycle, therefore the system and analysts can evaluate E at every cycle exactly.

Overwriting outputs can cause problems when a system reacts on the outputs immediately. The output variable, C in <Fig. 5>, which is evaluated from the top to the bottom, indicates the undependable case of overwriting a output. The C could have two different values, which are C:=A+B or C:=D−E+C, in one cycle. The overwriting can make a serious problem when the C immediately triggers a specific behavior of the system such as shutting down power-generating nuclear reactor.

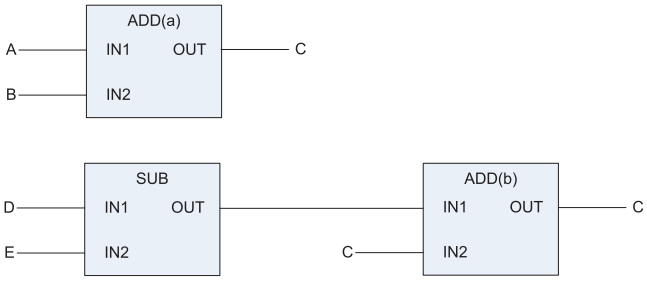

Fig. 5. Undependable case B: The undependable case about wrong usage of an output variable

Another important issue in <Fig. 5> is that input C of

ADD(b) is ambiguous whether it is a feedback of output C of ADD(b) or extension of output C of ADD(a). A feedback variable is the variable which is associated with a feedback path—is said to exist in a FBD when the output of a function block is used as the input of a function block which precedes it in a FBD program. IEC 61131-3 standard allows explicit and implicit loops as a feedback path both. A wire from a output variable to an input variable indicates in the case of an explicit loop. On the other hand, it is an implicit loop that there is a output variable and an input variable which have the same name in a FBD program. The input C of ADD(b) could be one of them or both of them, because it depends on the development environment. If there is not any overwriting, an input variable, which has the same name as an output variable, is implicitly a feedback.

To make the system clear, especially in the case of safety-critical systems, the system must not have ambiguities, and regulation should restrict language usage, which can cause ambiguities. We suggest a strong regulation of usage about output variables. First of all, all overwriting is denied in FBD programs. All outputs must assigned once in a FBD program. Next, all outputs must be explicit whether they are feedbacks or simple outputs.

<Fig. 6> and <Fig. 7> describe a replacement of <Fig. 5> to eliminate the ambiguity. <Fig. 6>, using *Connector & Continuation*, substitutes for the case that input C of ADD(b) is an extension of C of ADD(a) in <Fig. 5>. Connectors and continuations are elements which extend wires without storage of data or association with data elements defined in IEC 61131-3 standard. <Fig. 7>, using a prefix '**feedback_**' to feedback variable, substitutes for the case that input C of ADD(b) is a feedback variable of output C of ADD(b) in <Fig. 5>.

<Fig. 6> evaluates C as below:

```
-   C := (D - E) + (A + B)
```

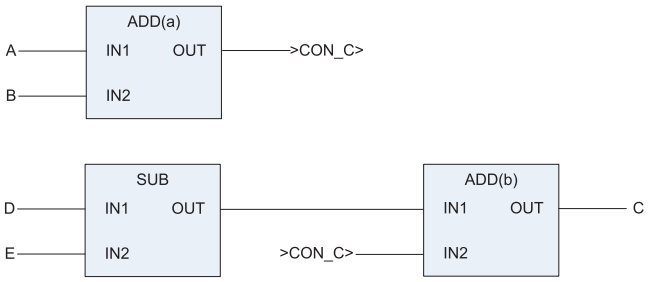

Fig. 6. Elimination of undependable case B using *connectors & continuations*

<Fig. 7> evaluates C and feedback_C as below

```
-   C := A + B
-   feedback_C := feedback_C + (D - E)
```

C. Guideline 3: Consensus of the Data Type

IEC 61131-3 standard said that functions or operations can be overloaded, which means it is able to operate on various types of input data within a generic type designator. <Fig. 8> shows that the function ADD is overloaded within two different data type; 1) INTEGER; and 2) BOOLEAN. Developers or

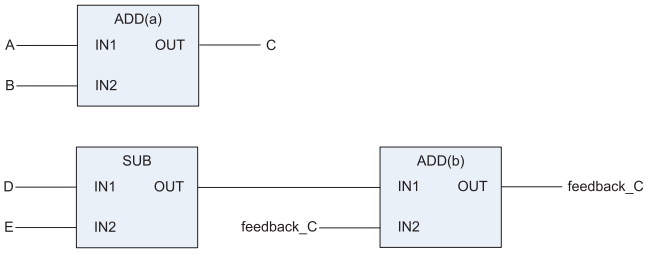

Fig. 7. Elimination of undependable case B using a prefix '*feedback_*' to feedback variable

analysts can evaluate the output, C, without confusion, because they might think that the function simply operates addition of two numbers. The target system, however, may not allow automatic type casting, which convert an expression of a given type into an another type.

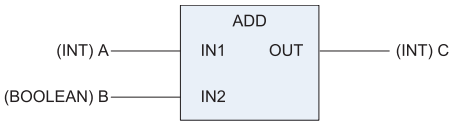

Fig. 8. Undependable case C: The undependable case not to correspond data types

There are type conversion blocks in IEC 61131-3 standard as described in <Fig. 9>, and it recommends that when all the formal input parameters of a standard function defined in the standard are of the same common type then all the actual parameters should be of the same type. This paper imposes a strict restriction on the consensus of the data type in FBD programs. All the standard blocks, which are able to have any number as inputs, must have a _[datatype] suffix to get rid of overloading. All the formal parameters and all the actual parameters, in addition, have to be the same type, if necessary, with use of the type conversion blocks. <Fig. 10> shows application the two restrictions to <Fig. 8>. All functions and function blocks in the former examples, <Fig. 5-7>, also should have suffixes '_[datatype]' in the same way.

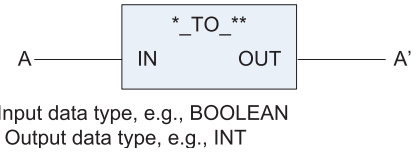

Fig. 9. The representative of type conversion functions

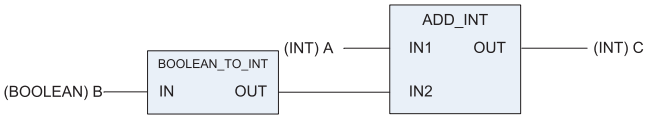

Fig. 10. Elimination of undependable case C using a type conversion function and a _INT suffix at a function ADD

D. Guideline 4: Initialization of Feedback Variables

Variables can be initialized by one of the mechanisms below:

- the default initial value(s) of the underlying elementary data types as defined in IEC 61131-3;
- NULL, if the variable is a reference;
- or the user-defined value(s) of the variable; this value is optionally specified in the variable declaration.

Explicit initiation of variables are not mandatory while initiation of feedback variables is essential. Feedback variables also can be initialized in the one of the mechanisms. When a system start operation, every input variables of its program receive data from other functions or function blocks, programs, or external devices (i.e., sensors, measuring instruments, etc.). The feedback variables, however, are not a variable which receives data from external though it is also used as an input variable. If developer does not define its initial value with its exact meaning, then the system may not operate as developer's intention. To prevent the undependable operation, feedback variables must have not only initial value but also clear meaning of the value.

E. Guideline 5: Explicit Order of Evaluation

IEC 61131-3 standard said that the order in which networks and their elements are evaluated follows the rules below:

- No element of a network shall be evaluated until the states of all of its inputs have been evaluated;
- the evaluation of a network element shall not be complete until the states of all of its outputs have been evaluated;
- the evaluation of a network is not complete until the outputs of all of its elements have been evaluated, even if the network contains one of the execution control elements—jump elements with label;
- or the order in which networks are evaluated shall conform to the provisions for the FBD language—the evaluation of a network shall be complete before starting the evaluation of another network which uses one or more of the outputs of the preceding evaluated network.

The order is not necessarily the same as the order, in which they are labeled or displayed, and it even does not have to be explicit. In <Fig. 11>, there are two function blocks which is labeled with its evaluation order above each function blocks. The labels seem as though the program will operate in ascending order, GE_INT is the first and ADD_INT is the second. On the contrary to this, <Fig. 11> operates in the opposite order, ADD_INT is the first and GE_INT is the second according to order the third rule.

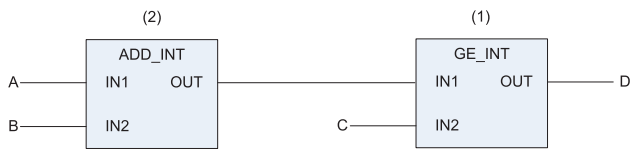

Fig. 11. Undependable case E: The undependable case of mislabeling order of the evaluation

The labeling like as <Fig. 11> is not a violation of the rules in the standard. It says the labeling is not necessary to be the same order in actual operations. Nevertheless, developers should avoid the mislabeling not to make analysis difficult and confused. One of two solutions are possible to eliminate the mislabeling:

1) programming without labeling order;
2) programming with labeling order in the same order of evaluation.

<Fig. 12> shows the elimination of the mislabeling in <Fig. 11> and correct labeling.

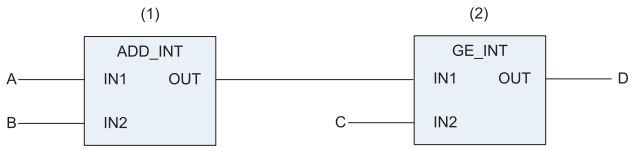

Fig. 12. Elimination of undependable case E labeling order in the same order of evaluation

IV. CASE STUDY

We applied the proposed practical guidelines about FBD programming to one of 18 shutdown logics, FIX_RISING, in the Bistable Processor (BP) program, which is a preliminary version of the Advanced Power Reactor's (ARP-1400) reactor protection system (RPS). The logic is developed using pSET [2]. Following subsection describes how the FIX_RISING logic operates and what elements are undependable. Next subsection describes a dependable version of FIX_RISING logic applying the guidelines.

A. FIX_RISING logic

<Fig. 13> shows a part of original FIX_RISING logic. The logic consists of two logics which evaluate trip signal (TRIP) and its pre-one (PTRIP). We only show a part of the logic related with the TRIP in the paper. There are three input variables, three output variables, and 13 functions and function blocks. TRIP_LOGIC is set to *TRUE* (1) when PV_OUT is higher than TSP over MAXCNT times in a row. It counts how many times PV_OUT is higher to TRIP_CNT. TSP is reset to TSP minus HYS when TRIP_LOGIC is set to *TRUE* until it returns *FALSE* (0).

The input variables—PV_OUT, MAXCNT, and HYS—are clear. All input variables have the same data type of corresponding functions. There are, however, ambiguity on the output variables—TRIP_CNT, TSP, and TRIP_LOGIC. PTRIP_CNT is in the upper right corner by SEL_DINT and in the middle of left by GE_DINT. The second PTRIP_CNT which is used as one input of GE_DINT is not certain whether it is a feedback variable or a extension of first one. Other outputs also have ambiguity in the same way.

The program may overwrite output values on two output variables, TRIP_LOGIC and TSP, twice in a cycle, because each of them has connections with two functions as an output. They may have two different values in a cycle, and they possibly cause unpredictable problems.

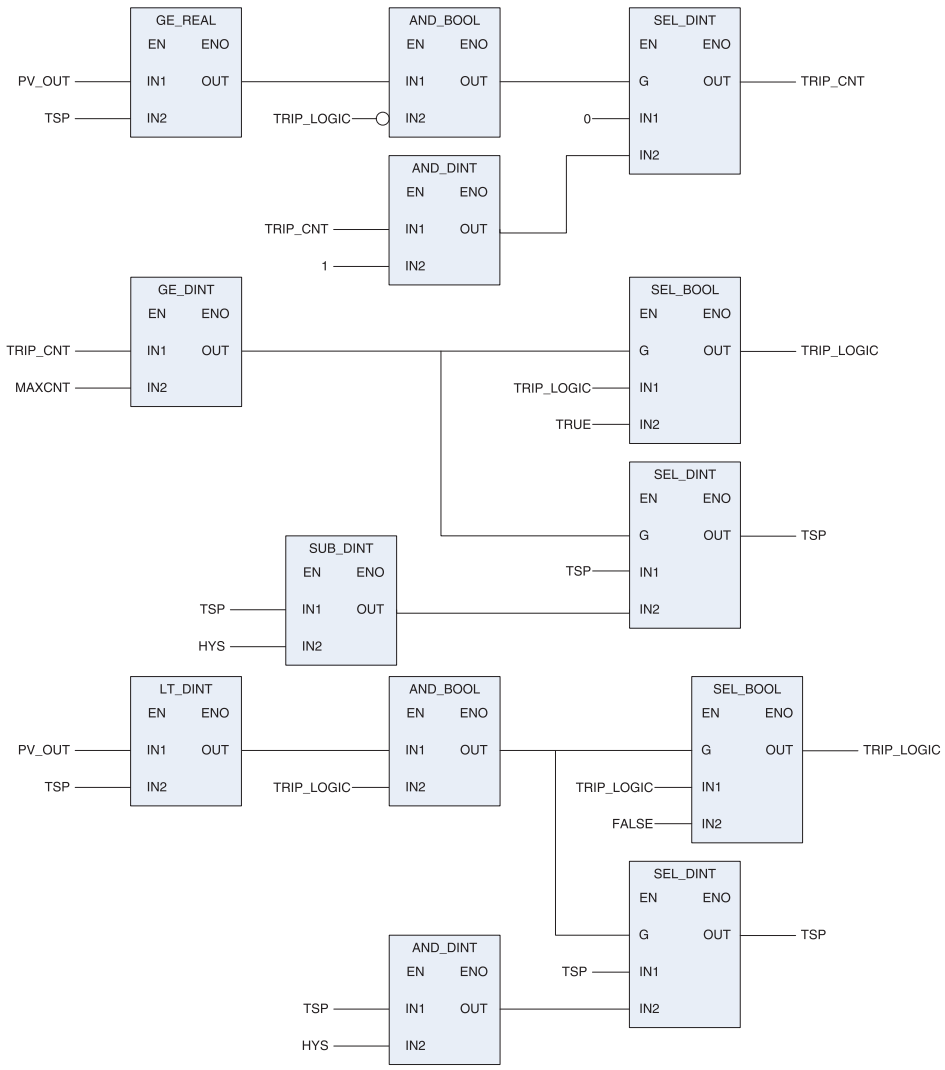

Fig. 13. A part of the original FIX_RISING logic related with the trip signal (TRIP)

The function blocks are not labels which express the order of evaluation. FBD programs developed using pSET are top-down programs which means that it evaluates the output from the top to the bottom. Although analysts can analyze the order of evaluation as the top-down program and the provisions for the FBD language in IEC 61131-3 standard, explicit order is useful for the analysis.

B. Application of the guidelines to the FBD program, FIX_RISING

We applied the 5 practical guidelines about FBD programming to <Fig. 13> to eliminate undependable elements. First, we labeled functions and function blocks with a number in parentheses (Guideline E). `GE_DINT` in <Fig. 14> is labeled `(1)` above. The labeling starts from left to right because the evaluation of the FBD program is completed before starting the evaluation of another network which uses one or more of the outputs of the preceding evaluated network. In addition, the evaluation order starts from the top because it is the top-down program developed using pSET.

Next, to eliminate ambiguity of usage of output variables, we used *connector & continuation* and a prefix `feedback_` to the variables (Guideline B). To apply the guideline B, we should know how pSET handles the output variables. pSET handles an output variables as followings:

- If a output variable is of the first time to use it as an input of function blocks in the order of evaluation, the output variable is a feedback variable.
- If a output variable is not of the first time, the output variable is an extension of a output variable which is the most recently used one as an output.

We classified the usage of output variables according the features and applied the guideline as describe in <Fig. 14>.

Application of the guidelines changes three output variables, and they are used 6 times as input variables as feedback variables. Black arrows in <Fig. 14> means the changes from output variables to feedback variables with prefix `feedback_` pointed by black arrows. Three uses of outputs are changed

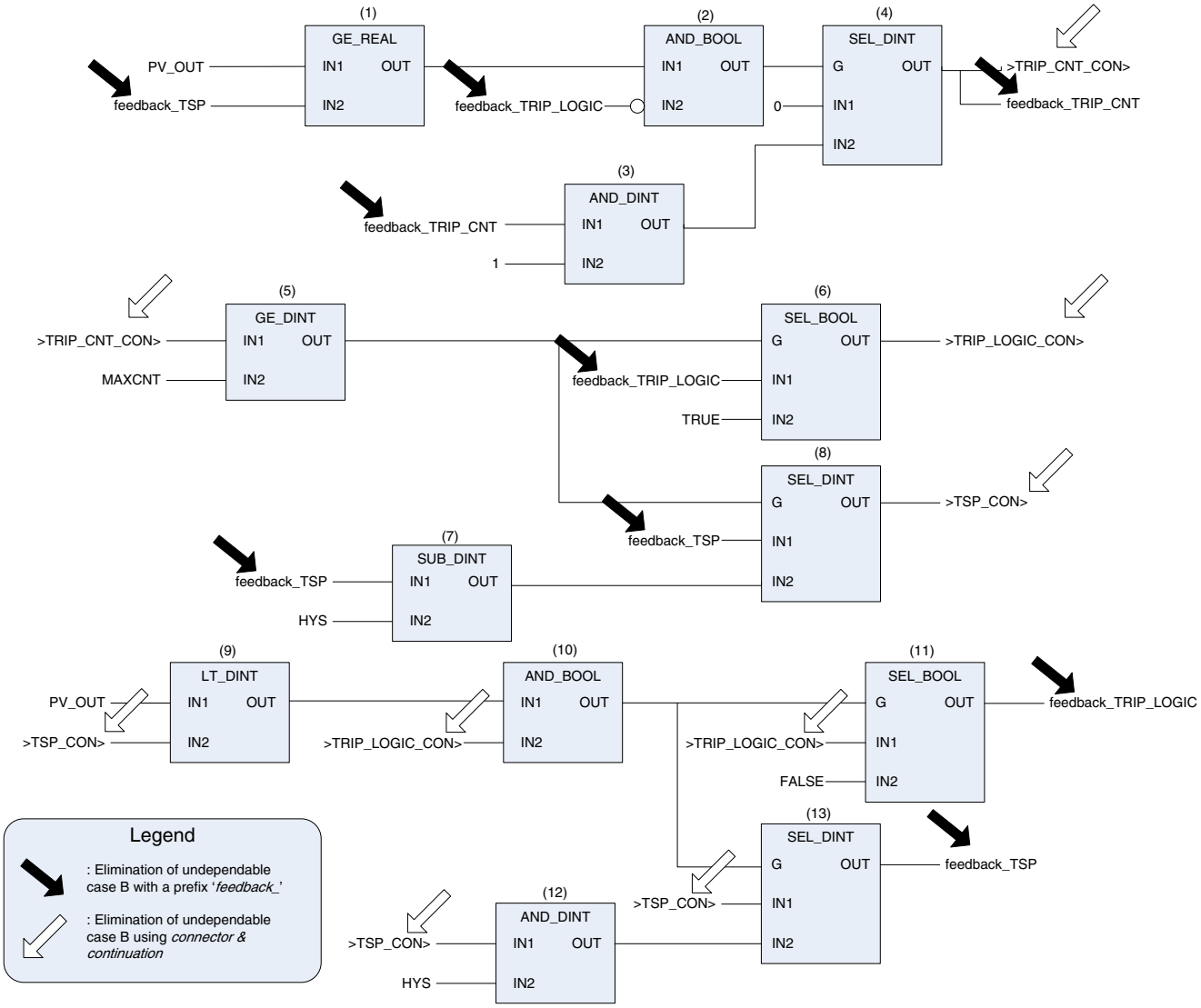

Fig. 14. FIX_RISING program applied guidelines

to three connectors and its extensions are changed to continuations pointed by white arrows. The outputs used as extensions must be substituted with the pairs of connectors and continuation as many as they used.

Name of variables could have an important meaning in safety-critical systems. In the case, a prefix or a suffix may give another meaning to developers or analysts. It is possible that a program has two output variables for purposes—one is only for a feedback variable, and another is only for an external output variable—which have the same value as described in <Fig. 15>.

A white arrow points a continuation, and a black arrow points a feedback variable. The two elements had the same name as `PTRIP_LOGIC`. To avoid ambiguity, application of guideline B changes the two elements. If it is necessary to use the original name, `PTRIP_LOGIC`, developers just can add a simple output variable presented in <Fig. 15>.

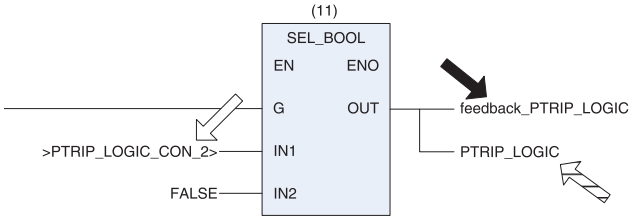

Fig. 15. Maintenance of a name of a specific output variable

C. Comparison between original FIX_RISING and modified FIX_RISING

Table I shows comparison of elements in the original FIX_RISING logic and modified FIX_RISING logic—<Fig. 13> and <Fig. 14>. Although modified one has more elements, feedbacks and connectors & continuations, the program

has no ambiguity which means all FBD programming tools which follows IEC 61131-3 standard are readable in the same way.

TABLE I. COMPARISON OF ELEMENTS IN THE ORIGINAL FIX_RISING AND THE MODIFIED FIX_RISING

	Original FIX_RISING	Safe FIX_RISING
# Blocks	13	13
# Input	3	3
# Output (Feedback)	3(0)	3(3)
# Conn. & Cont.	0	3

The original FIX_RISING is functionally correct in pSET. To confirm the functional equivalence between the original one and modified one, we analyzed its implementation program which is written in C language[2]. The orders of evaluation are same before and after. Noticeable changes by the guidelines are the assignment statements of the feedback variables. pSET generates temporal variables to deliver the output value to another function block—from `SEL_DINT(4)` to `GE_DINT(5)`—, because of use of connectors and continuations to eliminate the confusion about the extension or feedbacks. The program does not overwrite the output variable anymore during a cycle. Therefore, there are no risks about temporal output values. We identified that delivery of the value between two function blocks is functionally equivalent.

V. CONCLUSION AND FUTURE WORK

The paper introduces 5 undependable cases of FBD programming, and described that how they cause problems. To eliminate the undependable cases, we suggested the 5 guidelines about FBD programming. The proposed guidelines help analysts and developers evaluate the program without ambiguity or uncertainty such as meaning and usage of variables, order of evaluation, initialization, and etc. Furthermore, all FBD programming tools which follow IEC 61131-3 standard can handle the FBD program within the guidelines in the same way. When we applied the guidelines on a FBD program of the KNICS project, it changed several elements which have difficulty to be analyzed without specific knowledge of pSET. The results of the case study convincingly demonstrated the effectiveness of the proposed guidelines. We are now planning to develop more guidelines and an automatic tool which inspects FBD programs whether they follow the guidelines or not. The tool will obey PLCopen (*de facto standard*) [20] not to be dependent on specific vendors. We expect that the guidelines and tool help engineers implement more dependable programs.

ACKNOWLEDGMENT

This research was partially supported by 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, and also partially supported by a grant from the Korea Atomic Energy Research Institute, the development of the core software technologies of the integrated development environment for FPGA-based controllers.

[2]The C programs are automatically generated by a translator in the pSET. We omitted the programs in the paper because of the lack of space.

REFERENCES

[1] I. Sommerville, *Software engineering 8th edition: Chapter 20*, ser. International computer science series. Addison-Wesley, 2007.

[2] S. Cho, K. Koo, B. You, T.-W. Kim, T. Shim, and J. S. Lee, "Development of the loader software for plc programming," in *Proceedings of Conference of the Institute of Electronics Engineers of Korea*, vol. 30, no. 1, 2007, pp. 595–960.

[3] KNICS (Korea Nuclear Instrumentation and Control System R&D Center), http://www.knics.re.kr/english/eindex.html.

[4] KINS (Korea Institute of Nuclear Safety), http://www.kins.re.kr.

[5] *IEC 61131-3 International standard for programmable controllers - Part 3: Programming languages*, International Electrotechnical Commission, 1993.

[6] *IEEE Std 1364-2001: IEEE Standard Verilog Hardware Description Language*, Institute of Electrical and Electronics Engineers, 2001.

[7] D. Yoon, S. Hwang, K. Choi, and K. Park, *Implementation of C Code Generation Compiler Algorithm for IEC61131-3 Standard Language*, POSCON ICT, Nov 2005, http://rnd.poscon.co.kr/cyber/19-2.pdf.

[8] J. Yoo, S. Cha, and E. Jee, "Verification of plc programs written in fbd with vis," *Nuclear Engineering and Technology*, vol. 41, no. 1, pp. 79–90, Feb 2009.

[9] J. Yoo, J.-H. Lee, S. Jeong, and S. D. Cha, "Fbdtoverilog: A vendor-independent translation from fbds into verilog programs," in *SEKE*, 2011, pp. 48–51.

[10] *NUREG/CR-6463: Review guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems*, United States Nuclear Regulatory Commission, 1996.

[11] *Functional safety of electrical/electronic/programmable electronic safety-related systems: Part 3. Software requirements (IEC 61508-3)*, International Electrotechnical Commission, 1997.

[12] *Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178*, Radio Technical Commission for Aeronautics, 1992.

[13] *The international standard IEC 62304 ? medical device software ? software life cycle processes*, International Electrotechnical Commission, 2006.

[14] *Road vehicles – Functional safety: Part 6. Product development at the software level*, ISO, International Organization for Standardization, 2010.

[15] *Guidelines for the Use of the C Language in Critical Systems*, The Motor Industry Software Reliability Association, Oct 2004.

[16] *Guidelines for the Use of the C++ Language in Critical Systems*, The Motor Industry Software Reliability Association, Jun 2008.

[17] *Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions*, International Electrotechnical Commission, 2006.

[18] Liverpool Data Research Associates (LDRA) Testbed, http://www.ldra.com/misrac.asp.

[19] "Safety-critical application development environment (scade) suite," http://www.esterel-technologies.com/products/scade-suite/.

[20] PLCopen, http://www.plcopen.org/.