Programming Guidelines for FBD Programs in Reactor Protection System Software

Sejin Jung[a], Dong-Ah Lee[a], Eui-Sub Kim[a], Junbeom Yoo[a] and Jang-Soo Lee[b]
[a]*Division of Computer Science and Engineering College of Information and Communication, Konkuk University Seoul, Republic of Korea*
[b]*Man-Machine Interface System team Korea Atomic Energy Research Institute, Daejeon, Republic of Korea*
Corresponding author: jsjj0728@konkuk.ac.kr

1. Introduction

Safety of software in critical systems, such as nuclear power plants and vehicles, is one of the most important properties, because loss of the safety results in damages to the environment or human. Properties of programming languages, such as reliability, traceability, etc., play important roles in software development to improve safety. Several researches are proposed guidelines about programming to increase the dependability of software which is developed for safety critical systems [1,2]. Misra-c is a widely accepted programming guidelines for the C language especially in the sector of vehicle industry [3,4]. NUREG/CR-6463 helps engineers in nuclear industry develop software in nuclear power plant systems more dependably [5,6].

FBD (Function Block Diagram), which is one of programming languages defined in IEC 61131-3 standard [7], is often used for software development of PLC (programmable logic controllers) in nuclear power plants. Software development for critical systems using FBD needs strict guidelines, because FBD is a general language and has easily mistakable elements. There are researches about guidelines for IEC 61131-3 programming languages [8,9,10]. They, however, do not specify details about how to use languages.

This paper proposes new guidelines for the FBD based on NUREG/CR-6463. It not only provides elements usages as other guidelines for text languages, but also includes elements' placement because FBD is a graphical language. The paper introduces a CASE (Computer-Aided Software Engineering) tool to check FBD programs with the new guidelines and shows availability with a case study using a FBD program in a reactor protection system.

The paper is organized as follows. Section 2 describes the related work and Section 3 explains the new guidelines. Section 4 describes FBDChecker which is an automatic guideline checker, Section 5 explains a case study and finally Section 6 explains conclusion and future work.

2. Related work

2.1 Function Block Diagram

FBD is a graphic language based on blocks defined IEC 61131-3 standard programming languages [7]. The standard defines 10 categories and we present 6 out of the 10 in <Fig. 1>. The behavior of the blocks is intuitive as their names imply: ADD, AND, etc. Developers wire blocks from inputs to outputs in a manner similar to a circuit diagram makes to implement programs

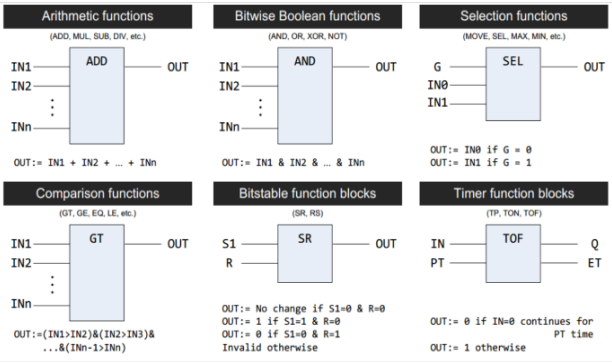

Fig 1.Examples of Function Blocks

2.2 Safe programming

Most errors and failures of software are results from human errors. Various standards restrict usages of languages to support software development for safety critical systems. For example, IEC 61508-3 has design and coding standards which are applied various fields [2]; Misra-c is a coding standard for software development in automotive industry [3,4]; DO-178B [11] is for airborne systems, and many other standards exist for other industry also.

NUREG/CR-6463 provides programming guidelines to develop software in nuclear domain. It supports not only IEC 61131-3 programming languages but also Ada, C/C++, Pascal, PL/M also [5,6].

PLCopen, as an organization active in Industrial Control, released technical specification about Safety Software for IEC 61131-3 standard [10]. This standard provides guidelines, and basic specifications of function blocks for implementation and use in safety-related environments. It helps a developers reduce effort to fulfill basic safety requirements like a distinction between safety and non-safety function, use of data type and so on.

3. Guidelines for FBD programming

NUREG/CR-6463 has 4 aspects—reliability, robustness, traceability, and maintainability—for safe programming. Guidelines in the reliability are to improve dependability and to guarantee correctness about simulation or action of a program. Guidelines in the maintainability increase readability and decrease

complexity. The robustness in the guidelines is for exception handling, and so on. Finally the traceability is 'use of built-in function,' 'use of compiled library' [5,6].

This paper proposes the new guidelines, which refine the NUREG/CR-6463 and suggest new guides for the FBD languages; they cover important properties for software safety which former guidelines do not include. The new guidelines consist of two properties, reliability and maintainability, because only the two of 4 properties in NUREG/CR-6463 are suitable for the FBD languages. Properties of robustness and traceability in NUREG/CR-6463 are excluded the new guidelines, because 'exception handling' and 'use of compiled library' are not acceptable in FBD languages.

3.1 Guidelines for reliability

Guidelines for the reliability consist of 4 categories as follows:

- Eliminating incorrect control flow
- Eliminating incorrect function uses
- Eliminating incorrect variable uses
- Eliminating explicit/implicit type conversion

Eliminating incorrect control flow If FBD programs have incorrect control flows, developers cannot predict its result. FBD programs, which developers are not able to predict, are not acceptable in safety critical systems. Because of unpredictable FBD programs are possible to make errors or not.

The category of eliminating incorrect control flow consists of rules to obtain dependability by modifying incorrect program flow. Rules are 'does not use incorrect explicit execution order', 'eliminating input port without connection any input', 'decreased using jump' and so on. If order in a program is incorrect, it is possible to make incorrect data flow and make incorrect output result.

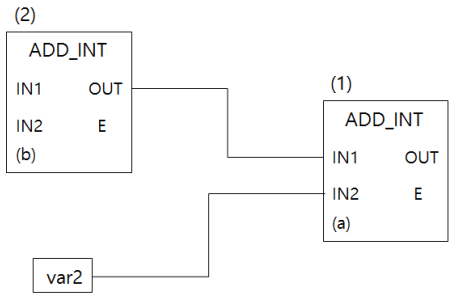

Fig 2. An example of incorrect execution order

For example, if a function 'a' uses a result value of a function 'b,' 'b' must execute before 'a.' if, however, developers assign execution order id of 'a' before 'b,' it is incorrect order and makes critical errors. <Fig.2> shows a diagram in this example. Using input port without connection is also a problem because a function which has useless input may not operate correctly.

Eliminating incorrect function/variable uses Using incorrect functions and variables are also problems. An incorrect function/variable makes incorrect results or meaningless values.

Guidelines in reliability about variables and functions include rules about correctness of types about function/variable. Guidelines also include rules are about 'initialization of variables,' 'do not use built-in function type which is different to the standard.' For example, according to a standard, ADD function can use INT and REAL types but if developers use other type, it may makes problems. And if developers use variable without initialization, result of this operation may be unexpected value. It may cause of errors or problems

Eliminating implicit/explicit type conversion A type conversion may make incorrect values or results, because a type conversion changes an interpretation method of bits in a variable. Software development for safety critical systems needs to reduce the use of the type conversions. There are two kinds of the conversions, explicit type conversions and implicit type conversions. The difference between a function type and an input type makes an implicit type conversion.

Implicit type conversions may make unintended result because results of conversions may not be predictable. We strongly limit the conversions between a type of a block and its input. We recommend using explicit type conversion. But several kinds of conversions make unintended results even though developers use explicit type conversion. Rules are conversion with between signed and unsigned, integer and real, different bits in same type, for example INT (4byte integer) to SINT (1byte integer), and so on. Type conversions, which are signed and unsigned, integer and Boolean, may change a variable's value because an interpretation of bits is different each other. As a result, software may operate in unexpected ways.

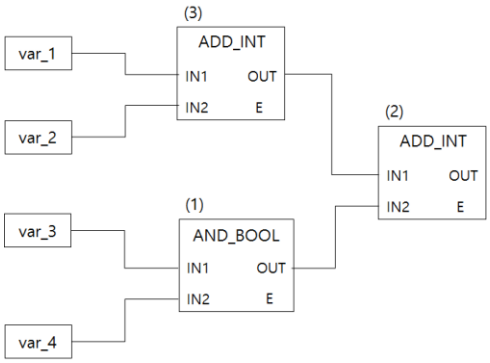

Fig 3. An example of violation about reliability rules

<Fig. 3> shows an example about violations of reliability rules. <Fig. 3> shows an implicit type conversion when *AND_BOOL* is connected input of *ADD_INT*. This is connections between functions with different types. It has problems in implicit type

conversion rules. And connections with two *ADD_INT* have execution order but its number is not correct. Because *ADD_INT* (2) executes later than *ADD_INT* (3). So this kind of ordering may be cause of problems. This reverse order should be changed. Programs, which are modified by applying guidelines, are shown in <Fig. 4>. It uses a type conversion function directly between *ADD_INT* and *AND_BOOL* and attaches correct execution order. However <Fig. 4> still has some problems about type conversion. Even though developers use an explicit type conversion function between a *BOOL* type and an *INT* type, it can make problems. So type conversion is not recommended for software of safety critical systems.

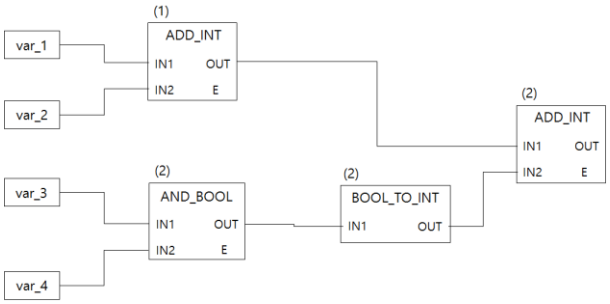

Fig 3. An example of modified in <Fig. 4>

3.2 Guidelines for maintainability

Guidelines for maintainability have two categories:

- Eliminating illegible diagram
- Eliminating illegible variable uses

An illegible diagram is difficult to modify and interpret it. So it may provide developers an opportunity of misinterpreting. Developers are able to make errors by a misinterpreting diagram and these errors make failures of software in systems. Therefore developers should eliminate an illegible diagram in software of safety critical systems. Rules in this chapter consist of reducing illegible in diagram.

Eliminating illegible diagrams The contents of this category are rules about drawing legible diagrams. Rules are 'lines or blocks are never crossed or overlapped each other,' 'restricting number of blocks in one user-defined function block,' and so on. Crossed lines or overlapped blocks are difficult to interpret diagram because it is not easy to find boundary of blocks. And we think that too many blocks in a user-defined function block are not easy to read also. A function block needs to limit a number of blocks to identify program flow easily.

Eliminating illegible variables uses The contents of this category are rules about blocks of a variable to identify easily. Rules are 'do not use a too short name or a too long name' and 'using an additional identifier if it is needed' and so on. A too short variable name is not easy to confirm a purpose of the variable and it decreases readability also. Using an additional identifier is a rule for identifying a variable easily. Feedback variables without an identifier sometimes make confusion of distinction between feedback variables and other variables. Feedback variables are which reuse by cycle of the program. <Fig. 5> shows an example about a feedback variable. A variable which is named *counter* is output of *ADD_INT* and input of *ADD_INT* at the cycle time. Depending on this characteristic, feedback variables have an effect in data flow of programs. So distinction of feedback variables is important.

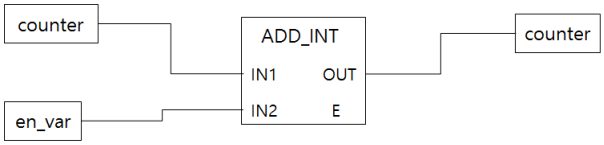

Fig 5. An example about feedback variable

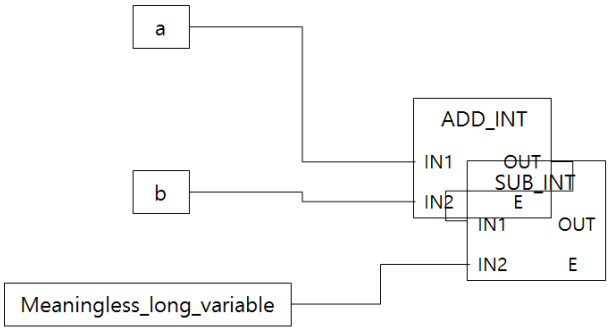

Fig 6. An example about violation of maintainability rule

<Fig. 6> shows an example about violations of maintainability rules. Lines, which connect *ADD_INT* and 'a,' 'b' each, are crossed. An overlap between two blocks, *ADD_INT* and *SUB_INT*, are not easy to read. Another problem in the example is too short name like an 'a,' and an example has a too long variable name also. Examples of explaining above reduce readability of FBD programs. Therefore we made rules in order to reduce these kinds of programming problems.

<Fig. 7> shows a modified figure by applying guidelines. We change variable names, which are 'a' and 'b.' And we remove crossed lines between connections with variables, 'a' and 'b.' Overlapping between *ADD_INT* and *SUB_INT* is removed from diagram also. So developers may understand <Fig. 7> easily rather than <Fig. 6>.

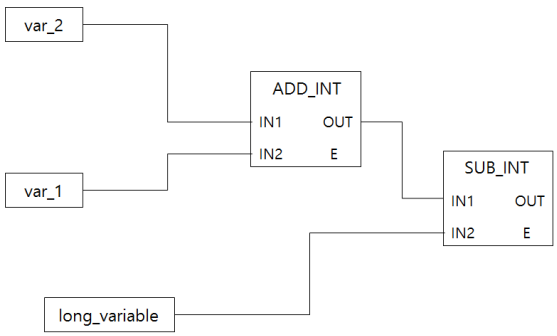

Fig 7. An example of modified in <Fig. 6>

3.2 Classification of the rules

We classify rules according to degree of affecting the FBD programs. Warning levels may affect a little or not, and error levels may make critical errors. <Table 1> shows description and an example about classification of the rules. The warning level is an illegible variable like a too short or a too long name, some of the explicit type casting, such as integer to integer, and so on. These kinds of violations are not made critical errors. And the error level is implicit type casting, incorrect control flow like incorrect execution ordering and so on. Error level violations should be removed from FBD programs.

Table I: levels in guidelines

level	Example
Warning	some of the explicit type casting, illegible variable, illegible diagram,
error	Implicit type casting, incorrect control flow, incorrect function/variable, some of the explicit type casting,

4. FBDChecker : An automatic rule checker for FBD programs

We developed a CASE (Computer-Aided Software Engineering) tool to check FBD programs for applying our guidelines easily. It detects violations automatically using FBD programs which are saved in a xml file. <Fig. 8> describes structure of the tool.

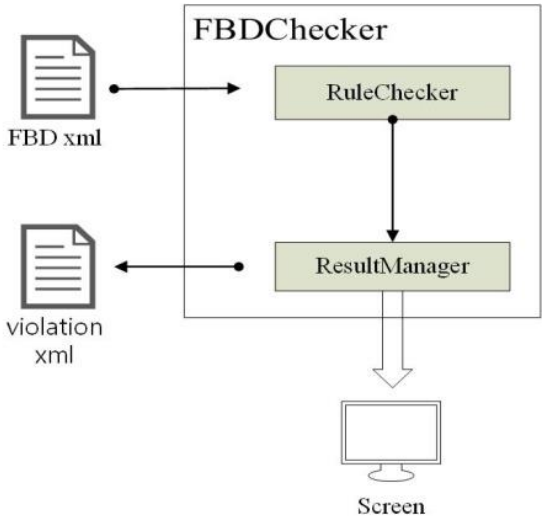

Fig 8. A structure of FBDChecker

FBDChecker consists of *RuleChecker* and *Result-Mager*. And an input and an output of FBDChecker is a file. Input file format of the tool is xml which is stored FBD programs by FBD xml schema of PLCopen TC6. PLCopen TC6 defined XML formats which is de facto standard for an interface between software tools using the IEC 61131-3 standard [12]. We also accept xml format for output because xml is a file which is used generally for sharing data. The file, which is used in output, format contains a data about information of a diagram and results of checking. FBDChecker reads a file, which is stored FBD programs, for using FBD programs to find violations. Next, RuleChecker finds violations and ResultManager combines the result and writes to file using JAXB library [13]. File format about result xml contains *pou* name, function position and name, *localId*, violation data, level.

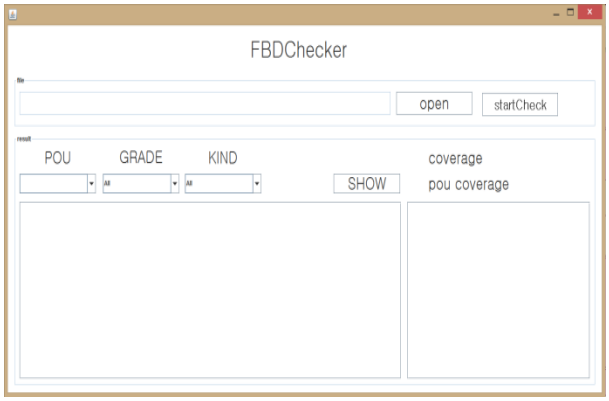

Fig 9. A screen dump of FBDChecker

<Fig 9> describes a screen of FBDChecker. It has an open button, a start check button, and a show button. A user clicks the open button in order to open a file and click start check to start finding violations. Show button is pressed, violation data appears to textbox in the bottom of the tool. Scroll boxes which named *pou*, grade and kind locate in the left side for data filtering. *Pou* means each function block name in FBD programs and grade means levels explain above—warning and error—, kind means blocks and variables. Blocks are function in FBD programs and variables are variables which are defined in FBD programs. FBDChecker shows filtered data which is selected by scroll box. In right side, FBDChecker shows the coverage about how much satisfy our guidelines.

5. Case study

We applied the proposed guidelines to a part of *FIX_RISING* which mentioned by [9] in the Bistable Processor (BP) program, which is a preliminary version of the Advanced Power Reactor's (ARP-1400) reactor protection system (RPS) by using FBDChecker. Thorough FBDChecker, finds 16 kinds of violations. Following figures describe violations that FBDChecker finds. <Fig. 9> and <Fig. 11> shows the results of FBDChecker. FBDChecker shows the list about function that violations are existed in the left side, and in the right side, shows the list about violations which are existed in the function.

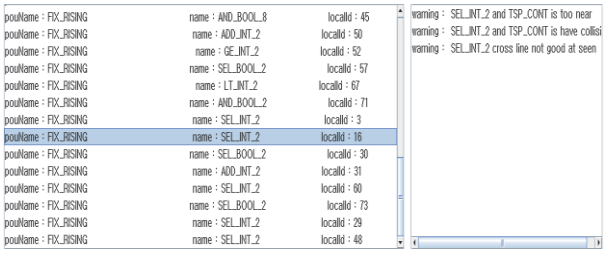

Fig 10. Examples of result about block in *FIX_RISING*

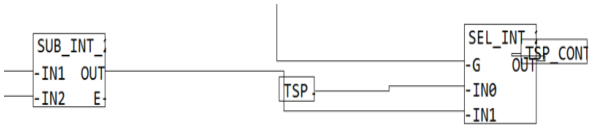

Fig 11. A diagram example of <Fig. 8>

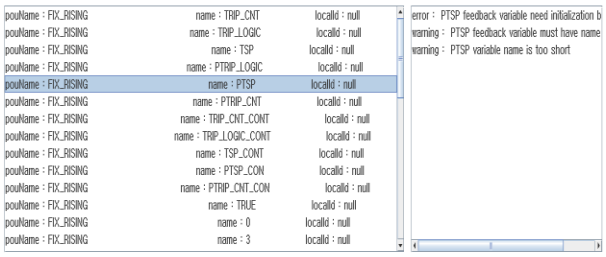

Figure 12. Examples of result about variable in *FIX_RISING*

Figure 13. An example of coverage in *FIX_RISING*

<Fig. 10> shows violations about 'collision (overlap) block,' 'too near block,' 'crossed line.' <Fig. 11> shows a part of diagram of *FIX_RISING* about <Fig. 10> [14]. The *SEL_INT* block overlaps the *TSP_CONT* variable. And line crosses the TSP block. Violations about using feedback variables without initialization and using too short name *PTSP* appears in <Fig. 12>. <Fig.13> shows an example of coverage in *FIX_RISING*. And it shows a coverage about each pou. FBDChecker finds violations which are a feedback variable without initialization, recommend attaching a feedback identifier, avoid using a short variable.

6. Conclusion and future work

In this paper, we suggested refined and new guidelines for development software in safety critical systems. And we proposed CASE tool for finding violations of rules. We expect that FBDChecker can help software development using the FBD language in safety critical systems. But several limits in rules about timer and some function in standard are existed.

We are now planning to evolve the tool, FBDChecker. It does not show a diagram directly inside the tool about violations. It is possible to check only the text to show violation. And we modify FBDChecker for expansion about rules later. Because of we modify source code directly to add new rules in FBDChecker. So it is difficult to add new rule. We reflect these kinds of plan and research more than it in the future.

Acknowledgements

This research was supported, in part, by a grant from the Korea Ministry of Science, ICT and Future Planning, under the development of the integrated framework of I&C dependability assessment, monitoring, and response for nuclear facilities. It was also supported, in part, by a grant from the Korea Atomic Energy Research Institute, under the development of the core software technologies of the integrated development environment for FPGA-based controllers.

REFERENCES

[1] D. Bonn, A. Canning, "SEMSPLC Guidelines for the Development of Safe PLC Application Software", IEEE Computing & Control Engineering Journal, June 1996, pp 141-143.
[2] Functional safety of electrical/electronic/programmable electronic safety-related systems: Part 3. Software requirements (IEC 61508-3), International Electrotechnical Commission, 1997.
[3] misra c : Guidelines for the Use of the C Language in Critical Systems, The Motor Industry Software Reliability Association, Oct 2004
[4] Guidelines for the Use of the C++ Language in Critical Systems, The Motor Industry Software Reliability Association, Jun 2008.
[5] NUREG/CR-6463 : Review guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems, United States Nuclear Regulatory Commission, 1997
[6] NUREG GUIDELINE FOR FBD, IEEE " http://grouper.ieee.org/groups/plv/HISTORICAL-LINKS/NUREG%20CR6463,%20Rev.%201/LANGUAGE/CH11FBD.HTM"
[7] IEC 61131-3 international standard part3 programming language
[8] D. Lee, J. Yoo, J. Lee, Guidelines for the Use of Function Block Diagram in Reactor Protection Systems
[9] Mario de Sousa, Restricting IEC 61131-3 programming languages for use on high integrity applications, emerging Technologies and Factory Automation 2008 p. 361-368 IEEE international conference on, 2008
[10] PLCopen, "Plcopen - technical committee 5: Safety software," Online publication, Jan 2006, http://www.plcopen.org/.
[11] software Considerations in Airborne Systems and Equip-ment Certification, RTCA-DO-178, Radio Technical Commission for Aeronautics, 1992.
[12] Plcopen, "http://www.plcopen.org/pages/tc6_xml/xml_-intro/" xml schema
[13] jaxb, JAXB Reference implementation, https://jaxb.java.net/
[14] D. Lee, E. Kim, Y. Seo, J. Yoo, FBDEditor : Design program of FBD for developing Reactor Instrumentation and Control system, Korea Conference on Software Engineering, P.315-318 , 2014.