Annals of Nuclear Energy 103 (2017) 454–469

Contents lists available at ScienceDirect

Annals of Nuclear Energy

journal homepage: www.elsevier.com/locate/anucene

A PLC platform-independent structural analysis on FBD programs for digital reactor protection systems ☆

Sejin Jung [a], Junbeom Yoo [a,*], Young-Jun Lee [b]

[a] Konkuk University, Republic of Korea
[b] Korea Atomic Energy Research Institute, Republic of Korea

A R T I C L E   I N F O

Article history:
Received 29 March 2016
Received in revised form 3 February 2017
Accepted 5 February 2017

Keywords:
Structural analysis
Function block diagram
PLC
Guidelines

A B S T R A C T

FBD (function block diagram) has been widely used to implement safety-critical software for PLC (programmable logic controller)-based digital nuclear reactor protection systems. The software should be developed strictly in accordance with safety programming guidelines such as NUREG/CR-6463. Software engineering tools of PLC vendors enable us to present structural analyses using FBD programs, but specific rules pertaining to the guidelines are enclosed within the commercial tools, and specific links to the guidelines are not clearly communicated. This paper proposes a set of rules on the structure of FBD programs in accordance with guidelines, and we develop an automatic analysis tool for FBD programs written in the PLCopen TC6 format. With the proposed tool, any FBD program that is transformed into an open format can be analyzed the PLC platform-independently. We consider a case study on FBD programs obtained from a preliminary version of a Korean nuclear power plant, and we demonstrate the effectiveness and potential of the proposed rules and analysis tool.

1. Introduction

PLCs (programmable logic controllers) have been widely used in the development of embedded controllers of various safety-critical systems. The software implemented with PLCs is typically programmed with PLC programming languages such as FBD (function block diagram), and then software engineering tools of PLC vendors synthesize the programs into PLC-executable codes mechanically. All issues related to structural correctness and the safety of the FBD programs need to be addressed in order to proceed with the mechanical synthesis. Software engineering tools developed by PLC vendors can check these issues strictly, but vendor-dependently and SW tool-internally.

RPS (Reactor Protection System) in nuclear power plants is implemented using PLCs, and the software used is often programmed with FBD. Because these systems are safety-critical systems, a hierarchy of standards, regulations, and guidelines on the structural quality (i.e., correctness and safety) of software programs should be satisfied to obtain operational approval of government authorities. IEC 61131-3 (IEC, 2013), IEC 61508 (IEC, 1997), and NUREG/CR-6463 (NRC, 1997) are some examples of such standards. NUREG/CR-6463 is top-most programming guideline for software development, and all software engineering tools developed by PLC vendors apply it in the analysis of FBD programs. In this paper, we focus on the problem whereby commercial tools perform the structural analysis well, but the exact correspondence (or relation) to upper rules and guidelines is neither clear nor opened.

In this paper, we propose a set of specific rules regarding the structure of FBD programs in accordance with the guidelines of higher levels. We can argue direct relations from one rule/guideline to upper ones. We also developed an automatic analysis tool "*FBD Checker*" for FBD programs in the PLCopen TC6 format (PLCopen XML schema Ver. 2.0, 2008). Any FBD program that is written or transformed into the open format can be analyzed the PLC platform[1]-independently. In other words, we do not have to use the software engineering tools of specific PLC vendors. We also performed case studies of structural analyses on FBD programs that were sampled from preliminary versions of Korean nuclear power plants. The results show the effectiveness and potential of the proposed rule sets and analysis tool.

The paper is organized as follows. In Section 2, we discuss background information such as FBD programming. Section 2 also

☆ This paper was originally published in *Korean Nuclear Society Autumn Meeting* 2014 (Jung et al., 2014).
* Corresponding author.
E-mail address: jbyoo@konkuk.ac.kr (J. Yoo).

[1] This paper uses the term '*PLC platform*' to indicate the pair (a PLC software engineering tool, a target PLC) such as (*pSET*, *POSAFE-Q PLC*) and (*SIMATIC-Manager*, SIMATIC Controller).

http://dx.doi.org/10.1016/j.anucene.2017.02.006
0306-4549/© 2017 Elsevier Ltd. All rights reserved.

describes a hierarchy of standards, rules, and guidelines for FBD programming, which are relevant to our discussion, i.e., developing PLC software in nuclear reactor protection systems. In Section 3, we explain detailed sets of guidelines that are proposed in this paper, along with examples, and in Section 4, we introduce the automatic structure analyzer - FBD Checker. We explain the case study in Section 5, while in Section 6, we conclude the paper and give remarks on our future research direction.

2. Background

2.1. FBD programming

FBD is one of the five PLC programming languages defined by the IEC 61131-3 standard (IEC, 2013), and it is the most widely used language to implement PLC-based safety-critical systems in nuclear power plants. FBD is a data-flow-based language that consists of function blocks that connect with each other. FBD programming is the process of connecting blocks to other blocks sequentially in order to produce appropriate outputs. Fig. 1 is an example of an FBD program, which is a part of '*fixed set-point rising trip*.' As we will explain in Section 5, this FBD program was originally an *NuSCR (nuclear software cost reduction)* (Yoo et al., 2005) formal requirements specification (KAERI, 2003), and the *NuDE (nuclear development environment)* framework (Yoo et al., 2014a; Yoo et al., 2014b; Kim et al., accepted) transformed it into a behaviorally equivalent FBD program.

FBD program in Fig. 1 consists of five function blocks, and it has a set of interconnections according to a predefined sequential execution order such as (27)–(31), which is labeled in Fig. 1. For example, the first function block that is executed is `LT_INT(27)`, while the last one is `AND_BOOL(31)`. `LT_INT` is a function block that calculates the logical '<' with two decimal integer inputs, and other function blocks can be understood in a similar way. The last function block `AND_BOOL` produces an output *TRIP*, which indicates a shutdown signal for nuclear reactors.

Fig. 2 shows a typical software development process for PLCs, used to develop safety-grade digital I&Cs. We first wrote the SRS (Software Requirements Specification) using in natural languages, and we then manually modeled the design specification using PLC programming languages such as FBD or LD. As commercial PLC vendors provide software engineering tools[2] to support mechanical translation from FBD/LD programs into C and executable codes for PLCs, most manual software programming will finish at the design phase. Structural analysis on FBD programs is also performed by PLC software engineering tools.

2.2. FBD Programming guidelines for safety systems in nuclear power plants

Fig. 3 summarizes FBD programming guidelines for safety systems in nuclear power plants, which are pertinent to our discussion - '*structural analysis on FBD programs*.' Below, the IEC 61131 Part 3 (IEC, 2013) defines 5 PLC programming languages, e.g., FBD, LD (Ladder Diagram) and ST (Structured Text), while Part 8 (IEC, 1993) provides basic programming guidelines to be followed with PLC programming languages. They define the FBD programming language and provide guidelines regarding how to program with FBD for PLC-based systems of general-purpose.

Based on the standards, the technical report of PLCopen TC5 (Safety Software Technical Specification, 2006; Safety Software Technical Specification, 2008) provides FBD programming guidelines for safety-critical systems. It suggests safe data types, which contain additional information for the safety status and level, as well as safe function blocks. PLCopen TC6 (PLCopen XML schema Ver. 2.0, 2008) also defines an open XML format for FBDs to enable the exchange of FBD programs with others, and this is because FBD programs produced by commercial software engineering tools of PLC vendors are not compatible with others. In this paper, we use the XML format to perform a structural analysis on FBD programs, vendor and tools (*i.e.,* the PLC platform-independently).

NUREG/CR-6463 (NRC, 1997; NRC, 1997) provides guidelines on software programming languages for nuclear power plant safety systems, as defined by the NRC (Nuclear Regulatory Commission) (NRC, 2015). It provides the following high-level languages, *e.g.,* Ada, C/C++, LD, FBD, Sequential Function Charts (SFC), Pascal, and PL/M. Further, it consists of 4 high-level categories such as *reliability*, *robustness*, *traceability*, and *maintainability*. The guidelines with respect to *reliability* are to improve the dependability and guarantee correctness, while the guidelines with respect to *maintainability* increase the readability and decrease the complexity. The *robustness* contains exception handling, and so on. The category of *reliability* also consists of 3 sub-chapters, namely *predictability of memory utilization*, *control flow*, and *timing*, and others also have several sub-chapters generically.

PLC vendors have provided safety-level PLCs and their own software engineering tools for developing safety-critical systems in nuclear power plants, as shown in Fig. 3. The commercial tools contain internal structural analysis facilities, which are internally referred to as the NUREG/CR-6463 guidelines. However, rules on FBD structures and specific mapping from the rules to the higher guidelines are not made public. While basic guidelines (SIEMENS PLC Control Systems, 2015; Invensys, 2006) with respect to how to program with function blocks have been publicized, specific rules in accordance with the higher guidelines have not been publicized.

2.3. Related work

To the best of our knowledge, few studies have focused on structural analysis using FBD programs. Lee et al. (2014) proposes 5 categories of FBD programming guidelines for safety-critical systems, such as data type, variable initialization, usage of variable, execution control, and explicit ordering. However, it does not clearly correspond to the top-most guideline, NUREG/CR-6463, whose target applications are safety systems in nuclear power plants. de Mario (2008) proposes 9 kinds of restrictions that should be followed to program with the IEC 61131-3 programming languages for high-integrity applications. Type safety, memory access, global variables, and conversion are some examples of the proposed restriction categories. As the restrictions are proposed for all PLC programming languages, e.g., LD and SFC, we need to select appropriate ones for FBD. The restrictions also need to be refined in detail to enable their use as rules for rule checking. For example, it deals with conversion between integers only.

Several researches that focus on static analysis, not rule checking on FBD programs, are as follows: Prahofer et al. (2012) provides 7 issues for static analysis on the IEC 61131-3 languages. Program complexity, unreachable codes, and performance problems are some examples. It also includes areas that are related to rule checking, such as naming convention. *Codesys* (CODESYS, 2015) is a programming tool of IEC 61131-3 programming languages that provides a static analysis on FBD programs. Using a technical data sheet (CODESYS, 2015), it also provides a list of subjects that includes useless declaration, detection of unreachable code, and naming convention. Fig. 4 depicts the above studies and our proposed approach – '*FBD Checker*' from the perspective of structural analysis on FBD programs.

[2] *e.g., 'TriStation 1131'* of *Invensys* (Invensys, 2015), '*SIMATIC-Manager*' of *Siemens* (SIEMENS, 2015), '*pSET*' of *PONU-Tech* (PONU-Tech, 2015; Cho et al., 2007), and '*SPACE*' of *AREVA* (AVERA, 2015).

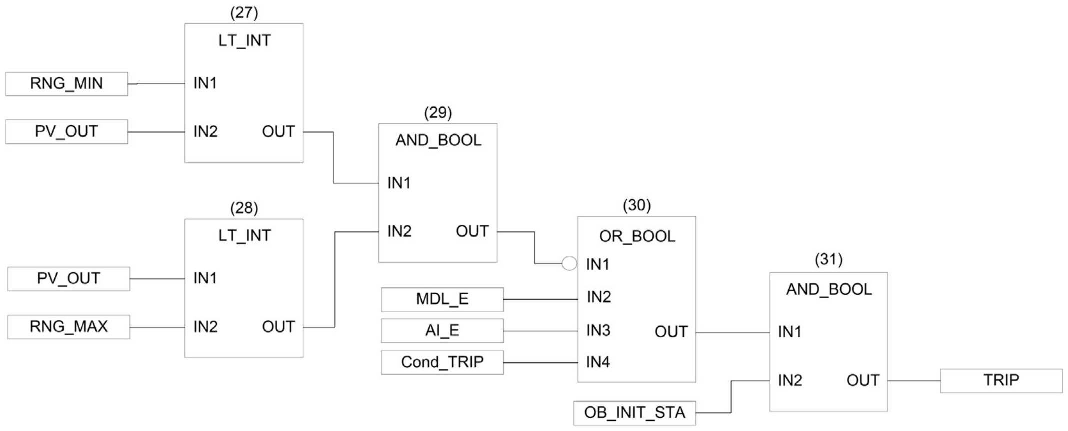


Fig. 1. An example of FBD for *fixed set-point rising trip*.

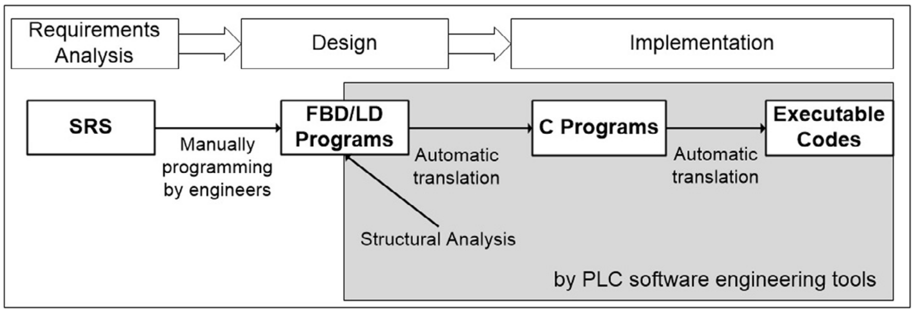


Fig. 2. Typical software development process for PLCs.

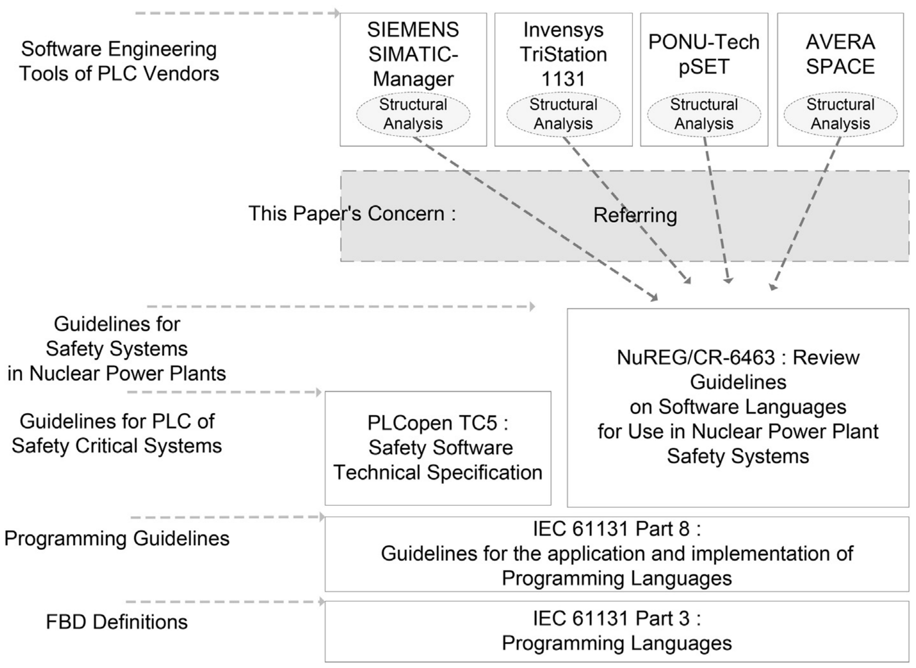

Software Engineering Tools of PLC Vendors
SIEMENS SIMATIC-Manager
Structural Analysis
Invensys TriStation 1131
Structural Analysis
PONU-Tech pSET
Structural Analysis
AVERA SPACE
Structural Analysis
This Paper's Concern :
Referring
Guidelines for Safety Systems in Nuclear Power Plants
NuREG/CR-6463 : Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems
Guidelines for PLC of Safety Critical Systems
PLCopen TC5 : Safety Software Technical Specification
Programming Guidelines
IEC 61131 Part 8 : Guidelines for the application and implementation of Programming Languages
FBD Definitions
IEC 61131 Part 3 : Programming Languages

Fig. 3. FBD programming guidelines.

We also performed to compare '*FBD Checker*' to top-most guideline and other related works, which we explained in the section above. Table 1 summarizes the correspondence from the top-most guideline NUREG/CR-6463 to each approach. Specific rules or guidelines are analyzed for every category of NUREG/CR-6463. According to the guidelines, the categories of robustness and traceability are excluded from the comparison because these are not appropriate for FBD programs. The table shows that '*FBD Checker*'

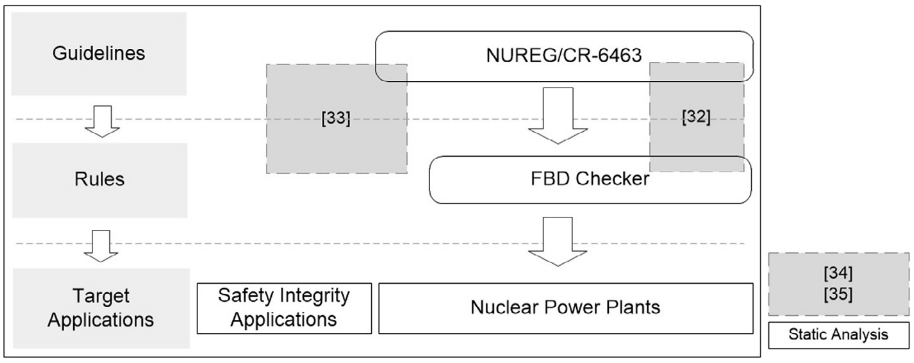


Fig. 4. An inclusion relation between related work and guidelines.

Table 1
Comparison viewpoint of structural analysis in the guidelines.

NUREG/CR-6463		FBD Checker	Lee et al. (2014)	de Mario (2008)	Prahofer et al. (2012)	CODESYS (2015)
Reliability	Maximizing Structure (1.2.1)	O	×	×	×	×
	Minimizing Control Flow Complexity (1.2.2)	O	×	Δ	Δ	O
	Variable Initialization (1.2.3)	O	Δ	×	×	Δ
	Single Entry and Exit Points (1.2.4)	O	×	×	×	×
	Interface Ambiguities (1.2.5)	O	×	×	×	×
	Data Typing (1.2.6), (1.2.7)	O	Δ	×	Δ	O
	Correct Ordering (1.2.8)	O	O	O	×	×
Robustness	Exception Handling (2.1)	–	–	–	–	–
Traceability	Use of built-in functions (3.1)	–	–	–	–	–
	Use of Compiled Libraries (3.2)	–	–	–	–	–
Maintainability	Drawing Diagram (4.1.1)	O	×	×	×	×
	Identifier Names (4.1.2)	O	Δ	×	O	×
	Mixed Language Programming (4.1.5)	O	×	×	×	×
	Minimize Use of Literals (4.1.8)	O	×	×	×	×
	Minimize Global Variables (4.2.3)	O	×	O	×	×
	Minimize interface complexity (4.2.4)	O	×	×	×	×

Table 2
Overview of the rules for structural analysis on FBD programming.

Category	The '*FBD Checker*'		NUREG/CR-6463	
	Sub Category	Number of Rules	Contents	Sub Category
1. Reliability	1.1 Correct Control Flow	1	1.2.1 Maximizing Structure	1.2 Predictability of Control Flow
		7	1.2.2 Minimizing Control Flow Complexity	
		1	1.2.8 Order of Precedence of Arithmetic, Logical and Function Operators	
	1.2 Correct Variables and Functions	2	1.2.3 Initialization of Variable before Use	
		1	1.2.4 Single Entry and Exit points in Subprograms	
		1	1.2.5 Minimization of Interface Ambiguities	
		6	1.2.6 Use of Data Typing	
		1	1.2.7 Precision and Accuracy	
	1.3 Type Conversion	1	1.2.6 Use of Data Typing	
		18	1.2.7 Precision and Accuracy	
	N/A	–	1.1, 1.2.9 ~ 1.2.13, 1.3	
2. Robustness	N/A	–	2.1.1, 2.1.2	2.1 Exception Handling
3. Traceability	N/A	–	–	3.1, 3.2
4. Maintainability	2.1 Drawing Diagrams	8	4.1.1 Layout of FBD Diagram	4.1 Readability
		1	4.1.5 Minimize Mixed Language Programming	
	2.2 Defining Variables	5	4.1.2 Descriptive Identifier Names	
		1	4.1.8 Minimize Use of Literals	
	2.3 Abstraction	1	4.2.3 Minimization of the Use of Global Variables	4.2 Abstraction
		1	4.2.4 Minimization of Interface Complexity	
	N/A	-	4.1.3, 4.1.4, 4.1.6, 4.1.7, 4.2.1, 4.1.2, 4.3 ~ 4.5	
	Total	56		

*N/A from NUREG/CR-6463: 1.1, 1.2.9, 1.2.11, 1.2.12, 1.2.13, 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5, 1.3.6, 4.1.4, 4.1.6, 4.1.7, 4.2.1, 4.2.2, 4.5.

can handle all categories sufficiently from the perspective of structural rule checking of FBD programs for safety systems in nuclear power plants.

3. Structural analysis rules for FBD programs

In this paper, we propose a set of specific structural analysis rules for FBD programs, which can bridge the gap between the top-level guideline NUREG/CR-6463 and FBD programs under development. The rules are all mapped to corresponding guidelines of NUREG/CR-6463, and are detailed enough for direct application into FBD programs. The rules are also implemented into an automatic checking tool – '*FBD Checker*'.

3.1. Organization of the rules

Table 2 provides a brief overview of the organization of the rules that are proposed in this paper, and which were implemented into the case tool '*FBD Checker*.' There are 56 rules with 6 sub-categories, and they all correspond to specific sub-categories of NUREG/CR-6463. For example, 11 rules of "*1.2 Correct Variables and Functions*" of *FBD Checker* are mapped to 5 sub-categories of "*1.2 Predictability of Control Flow*" in NUREG/CR-6463. As NUREG/CR-6463 defines that the categories of "*2. Robustness*" and "*3. Traceability*" and their 16 sub-categories are not applicable to FBD programs, we provide no rules for the categories. We also have 8 different *no-rules* (N/A), whereas NUREG/CR-6463 does not, because the technique of rule checking cannot handle them. "*1.2.10 Proper Handling of Program Instrument*" is an example of these cases. In summary, the 14 guidelines of NUREG/CR-6463 have been refined into 56 rules, which can be applied into FBD programs PLC platform-independently.

3.2. Rules of reliability

The category of "*1. Reliability*" consists of 39 rules with 3 sub-categories, as presented in Table 2. This subsection explains the rules according to their categories. Because of space limitations, we focus on new rules that are not covered by NUREG/CR-6463.

3.2.1. Correct control flow

For the correct execution safety of FBD applications, it is important to ensure that an FBD program is executed in accordance with a predefined execution order of function blocks. Any possible occurrence that may distort the execution order from the expectation of programmers should be avoided beforehand. Here, we propose 9 rules for detecting and avoiding incorrect control flow from FBD programs, and explanations on the new ones are as follows:

1.1 Correct Control Flow
1.1.1 The `JUMP` function block should be used less than MAX_JUMP times within a program
1.1.2 The number of function blocks in an FBD should be less than MAX_FUNCTION
1.1.3 All input ports in a function block should be connected to others
1.1.4 The type of a function block and that of its connected output variable should be the same
1.1.5 The `MOVE` function block should be used intentionally
1.1.6 All connections should be identified in advance
1.1.7 A predefined execution order should coincide with the connection of blocks
1.1.8 All variables should be used
1.1.9 An assignment should be used with a `MOVE` function block

Rule 1.1.3 All input ports in a function block should be connected to others

Fig. 5 shows an example in which an input port `IN1` of `AND_BOOL` has no connections with others. As commercial PLC software engineering tools provide the input port with an initial/default value (*e.g., 0*) implicitly, programmers may take no note of it.

Rule 1.1.4 The type of a function block and that of its connected output variable should be the same

Fig. 5 also shows an example where the type mismatch results in an implicit-type conversion. While `OR_BOOL` produces a Boolean output, it has been converted into an integer *Result_INT* implicitly.

Rule 1.1.5 The `MOVE` function block should be used intentionally

The `MOVE` function block is often used to deliver a variable to others through renaming. Fig. 6 illustrates the two cases. While `MOVE_INT` renames the input variable *Var1_INT* to *Result1_INT* intentionally, the two `MOVE_BOOL` have no effect on the execution result.

Rule 1.1.6 All connections should be identified in advance

Connection is widely used to make the FBD execution more obvious and clearer. If an output variable is subsequently used as an input, the connection (connector + continuation) can remove the connected (long) wires neatly, as depicted in Fig. 7. *Connection_INT* from `ADD_INT(2)` is subsequently used as an input to `SUB_INT(10)` and `MUL_INT(20)`. All connections in an FBD should be identified in advance in order to distinguish them from implicit feedback variables (IEC, 2013).

3.2.2. Correct variables and functions

To realize the safety of FBD applications, it is important to ensure that all I/O variables and function blocks are defined and used correctly according to the standard and guidelines. We propose 11 rules for detecting and avoiding incorrect uses of variables and functions, and for some of them, we provide detailed explanations.

1.2 Correct Variables and Functions
1.2.1 All local variables should be initialized
1.2.2 All feedback variables should be initialized
1.2.3 The number of output variables that are not feedback variables should be less than MAX_OUTPUT
1.2.4 The number of input types of user-defined function blocks should be less than MAX_MIXEDTYPE
1.2.5 The variable type should be defined
1.2.6 The function block type should be defined
1.2.7 Arithmetic function blocks should be used with the *ANY_NUM* type
1.2.8 Bitwise function blocks should be used with the *ANY_BIT* type
1.2.9 Timer function blocks should be used the *ANY_TIME* type
1.2.10 If it is available, use the *LREAL* type rather than *REAL*
1.2.11 '0' should not be used in `AND`

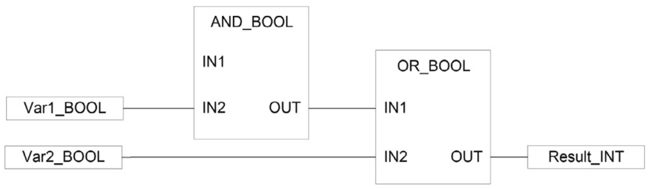


Fig. 5. An example of an input port with no connection.

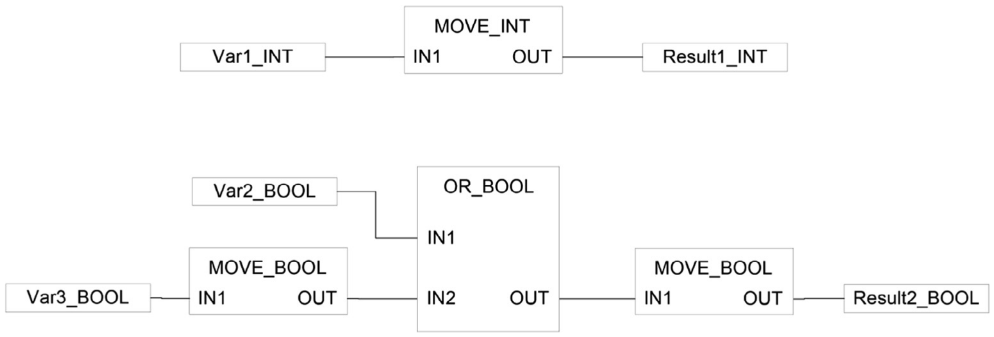


Fig. 6. The use of `MOVE` function block.

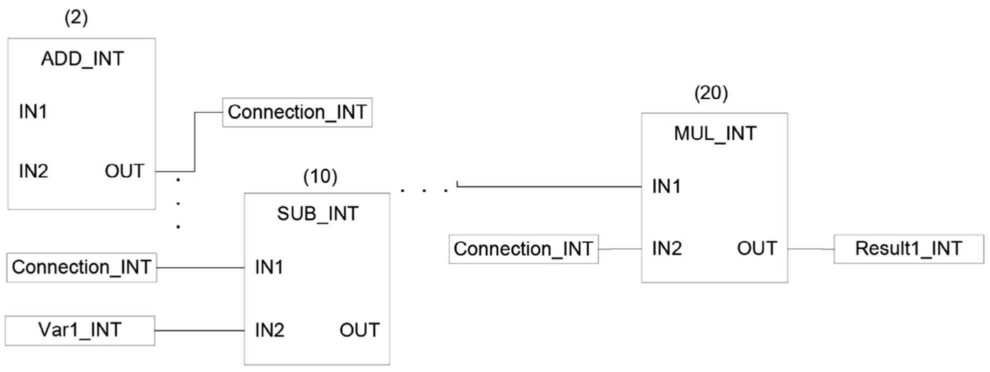


Fig. 7. An example of connections.

Rule 1.2.1 1.2.2 All local and feedback variables should be initialized

All variables that are not I/O variables as well as feedback ones, should be initialized before use, even if commercial software engineering tools initialize them with default values. Unintended execution may result from garbage values read from the undefined variables.

Rule 1.2.6 The function block type should be defined

Fig. 8 shows an example of using a type-undefined function block `ADD`. When the type of the function block is not defined, all types of input ports are possible, and the calculation may result in different types of output, *i.e.*, integer or real. Consequently, the next function block `ADD_INT` may produce incorrect outputs.

1.2.7 1.2.8 1.2.9 All built-in function types should be used with designated types

All built-in functions that are defined by IEC 61131-3 should be used with the designated types only. All arithmetic functions, such as `ADD` and `MUL`, should be defined with the *ANY_NUM* types. All bitwise functions, such as `AND` and `OR`, should be defined with the *ANY_BIT* types. On the other hand, selection and comparison functions can be defined with the *ANY_ELEMENTARY* type, as defined in ⟨Table 3⟩

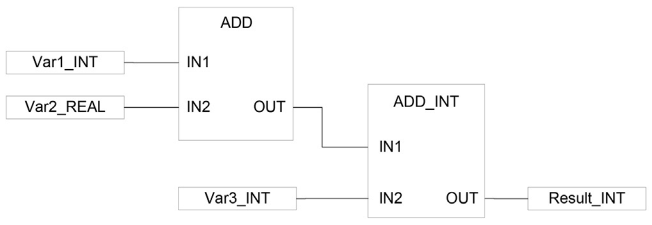


Fig. 8. An example of overload (undefined) function block.

3.2.3. Type conversion

A function block can be interpreted as different types in two ways, such as implicit- and explicit-type conversions. An implicit-type conversion is a potential source of unexpected and distorted control flow and data, and we have to use it carefully. For example, the output *REAL* value of `SEL_REAL` in Fig. 9 is converted into an *INT* value implicitly. The up-to-date IEC 61131-3 (IEC, 2013) also emphasizes the importance of type conversions. In accordance with the recommendation of NUREG/CR-6463, for all cases, we strictly prohibit the implicit-type conversion and propose the use of the explicit-type conversion. Some explicit-type conversions are also restricted to prevent loss of data.

Table 3 presents a hierarchy of the generic data types in IEC 61131-3. In this paper, we use the above-mentioned naming convention in order to avoid any unnecessary confusion. We also use (Other_Types) to indicate different types from specific one. We propose 19 rules regarding the type conversion. Implicit-type conversion should not be used, and programmers have to use explicit-type conversions appropriately. "*(A)* ⇒ *(B)*" indicates an explicit-type conversion from the data type *(A)* to the type *(B)*.

1.3 Type Conversion

1.3.1 Implicit-type conversion should not be used

1.3.2 Explicit-type conversion: (ANY_SIGNED) ⇒ (Other_Types)
  1.3.2.1 (Other_Types): (ANY_DURATION, ANY_DATE, ANY_CHARS) are not allowed
  1.3.2.2 (Other_Types): (ANY_UNSIGNED) is not allowed
  1.3.2.3 (Other_Types): (ANY_SIGNED) is allowed only with the 5 cases:
    ((SINT)⇒(INT, DINT, LINT), (INT)⇒(DINT, LINT), (DINT)⇒(LINT))

(*continued on next page*)

(continued)

1.3 Type Conversion

1.3.2.4 (Other_Types): (ANY_REAL) is allowed only with the 5 cases:
((SINT)⇒(REAL,LREAL), (INT)⇒(REAL,LREAL), (DINT)⇒(LREAL)))
1.3.2.5 (Other_Types): (ANY_BIT) is allowed only with the 4 cases:
((SINT)⇒(BYTE), (INT)⇒(WORD), (DINT)⇒(DWORD), (LINT)⇒(LWORD))

1.3.3 Explicit-type conversion: (ANY_UNSIGNED) ⇒ (Other_Types)
1.3.3.1 (Other_Types): (ANY_DURATION, ANY_DATE, ANY_CHARS) are not allowed
1.3.3.2 (Other_Types): (ANY_UNSIGNED) is allowed only with the 6 cases:
((USINT)⇒(UINT, UDINT, ULINT), (UINT)⇒(UDINT, ULINT), (UDINT)⇒(ULINT))
1.3.3.3 (Other_Types): (ANY_SIGNED) is allowed only with the 6 cases:
((USINT)⇒(INT, DINT, LINT), (UINT)⇒(DINT, LINT), (UDINT)⇒(LINT))
1.3.3.4 (Other_Types): (ANY_REAL) is allowed only with the 5 cases:
((USINT)⇒(REAL,LREAL), (UINT)⇒(REAL,LREAL), (UDINT)⇒(LREAL))
1.3.3.5 (Other_Types): (ANY_BIT) is allowed only with the 4 cases:
((USINT)⇒(BYTE), (UINT)⇒(WORD), (UDINT)⇒(DWORD), (ULINT)⇒(LWORD))

1.3.4 Explicit-type conversion: (ANY_BIT)⇒(Other_Types)
1.3.4.1 (Other_Types): (ANY_DURATION, ANY_DATE, ANY_CHARS) are not allowed
1.3.4.2 (Other_Types): (ANY_REAL) is not allowed
1.3.4.3 (Other_Types): (ANY_BIT) is allowed with the 10 cases:
((BOOL)⇒(BYTE, WORD, DWORD, LWORD), (BYTE)⇒(WORD, DWORD, LWORD)
(WORD)⇒(DWORD, LOWRD), (DWORD)⇒(LOWRD))
1.3.4.4 (Other_Types): (ANY_SIGNED) is allowed only with the 9 cases:
((BOOL)⇒(SINT, INT, DINT, LINT), (BYTE)⇒(SINT), (WORD)⇒(INT)
(DWORD)⇒(DINT), (LWORD)⇒(LINT))
1.3.4.5 (Other_Types): (ANY_UNSIGNED) is allowed only with the 9 cases:
((BOOL)⇒(USINT, UINT, UDINT, ULINT), (BYTE)⇒(USINT), (WORD)⇒(UINT)
(DWORD)⇒(UDINT), (LWORD)⇒(ULINT))

1.3.5 Explicit-type conversion: (ANY_REAL) ⇒ (Other_Types)
1.3.5.1 (REAL)⇒(LREAL) is only allowed

1.3.6 Explicit-type conversion: (ANY_DATE) ⇒ (Other_Types)
1.3.6.1 (LTOD)⇒(TOD) is only allowed

1.3.7 Explicit-type conversion: (ANY_CHARS)⇒(ANY_CHARS) is only allowed

Rule 1.3.1 Implicit-type conversion should not be used
An implicit-type conversion occurs when two variables or ports with different types are connected. For example, `ADD_INT` in Fig. 9 performs the addition of two values - an integer value from `IN2` and a real value from `IN1`, and then the result `OUT` is produced as an integer type. The result of `SEL_REAL` may lose the value below the decimal point implicitly.

Rule 1.3.2 Explicit-type conversion: (ANY_SIGNED) ⇒ (Other_Types)
The explicit-type conversion from (ANY_SIGNED) into (Other_Types) can be categorized as 5 types of (Other_Types). The example in Fig. 10 shows the explicit conversions from (INT)⇒(DINT), (DINT)⇒(LREAL) and (SINT)⇒(LREAL).

Rule 1.3.3 Explicit-type conversion: (ANY_UNSIGNED) ⇒ (Other_Types)
The explicit-type conversion from (ANY_UNSIGNED) into (Other_Types) can be categorized as 5 types of (Other_Types). The example in Fig. 11 shows the explicit conversions from (UINT)⇒(UDINT) and (UDINT)⇒(LREAL).

Rule 1.3.4 Explicit-type conversion: (ANY_BIT) ⇒ (Other_Types)
The explicit-type conversion from (ANY_BIT) into (Other_Types) can be categorized as 5 types of (Other_Types). The example in Fig. 12 shows the explicit conversions from (BOOL)⇒(WORD), (BOOL)⇒(INT) and (WORD)⇒(INT).

3.3. Rules of maintainability

The category "*2. Maintainability*" consists of 17 rules with three sub-categories, as presented in Table 2. The rules in this category focus on improving the maintainability of FBD programs by drawing diagrams neatly and using variables consistently.

3.3.1. Drawing diagrams

It is important to draw a function block diagram neatly from the perspective of readability. Even if it is correct and valid, overlapping diagrams or wires (lines) may be contributing factors to misunderstanding, and should be checked rigorously. We provide 9 rules of "*Drawing Diagrams*" as follows:

2.1 Drawing Diagrams

2.1.1 Lines should not be crossed
2.1.2 Blocks should not overlap
2.1.3 Lines should not be overlapped with blocks
2.1.4 Lines should not overlap
2.1.5 Blocks should not be within MIN_DIS pixel points of each other
2.1.6 Connected blocks should not be more than MAX_DIS pixel points apart
2.1.7 Customized ports in user-defined function blocks should not overlap
2.1.8 The output of a function block should be located at the right-hand side of the block
2.1.9 Other PLC languages such as LD and ST cannot be used in FBDs

Rule 2.1.1∼2.1.4 Lines and blocks should not cross or overlap
All crossed or overlapping lines and blocks should be detected and modified appropriately, as depicted in Fig. 13. The reason for this is that they make it difficult for programmers to understand the diagram clearly.

Rule 2.1.7 Customized ports in user-defined function blocks should not overlap
User-defined function blocks have customized names of input/output ports. These port names may be overlapped, as shown in Fig. 14. The output port `f_HI_LOG_POWER_Ptrp_Out` overlaps with the first input port, and it makes it difficult to understand the block clearly.

Rule 2.1.8 The output of a block should be located on the right-hand side of the block

Table 3
Hierarchy of the generic data types in IEC 61131-3)(IEC, 2013).

Generic data types					Generic data types	Groups of elementary data types
ANY						
	ANY_DERIVED					
	ANY_ELEMENTARY					
		ANY_MAGNITUDE				
			ANY_NUM			
				ANY_REAL		REAL, LREAL
				ANY_INT	ANY_UNSIGNED	USINT, UINT, UDINT, ULINT
					ANY_SIGNED	SINT, INT, DINT, LINT
			ANY_DURATION			TIME, LTIME
		ANY_BIT				BOOL, BYTE, WORD, DWROD, LWORD
		ANY_CHARS				
			ANY_STRING			STRING, WSTRING
			ANY_CHAR			CHAR, WCHAR
		ANY_DATE				DATE_AND_TIME, LDT, DATE, TIME_OF_DAY, LTOD

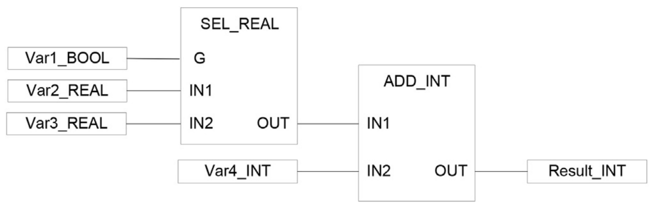


Fig. 9. An example of an implicit-type conversion.

An output variable of a function block (*i.e.,* an FBD) should be located on the right-hand side of the function block (or FBD). If not, as described in Fig. 15., it may result in a misunderstanding of FBDs.

3.3.2. Defining variables

The appropriate definition and use of variables are important to improve the maintainability of FBD programs. The size of variables and the use of additional identifiers are the concerns of this category. In the case of user-defined function blocks, local variables and global variables should be distinguished clearly. We propose the following 6 rules:

2.2 Defining Variables

2.2.1 Variable names should have more than 6 characters
2.2.2 Variable names should have less than MAX_NAME characters
2.2.3 Literals should not be used as variables
2.2.4 Global variables should be accompanied with additional identifiers
2.2.5 Feedback variables should be accompanied with additional identifiers
2.2.6 Connector/Continuation should be accompanied with additional identifiers

2.2.3 Literals should not be used as variables

A literal refers to an input/output variable that has a constant value, such as 0, 1, and 100. Literals are very widely used in programming FBDs. However, they may reduce the understandability of programs, and it is better to change them using a (constant) vari-

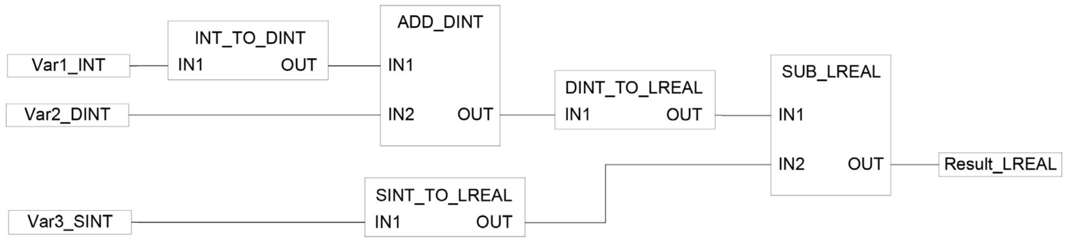


Fig. 10. An example of explicit-type conversions from *ANY_SIGNED*.

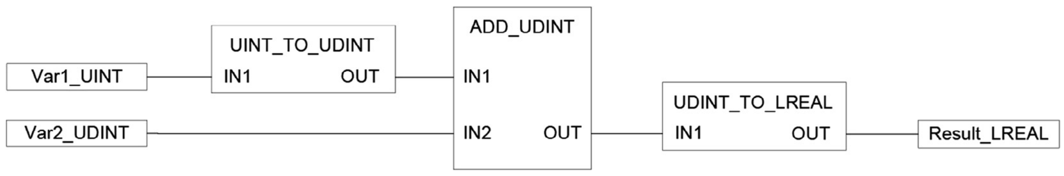


Fig. 11. An example of explicit-type conversions from *ANY_UNSIGNED*.

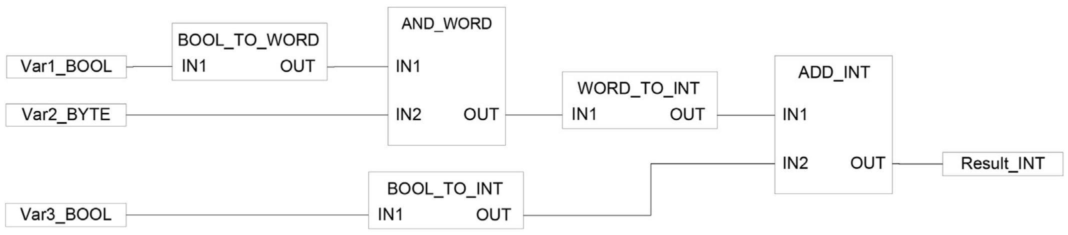


Fig. 12. An example of explicit.-type conversions from *ANY_BIT*.

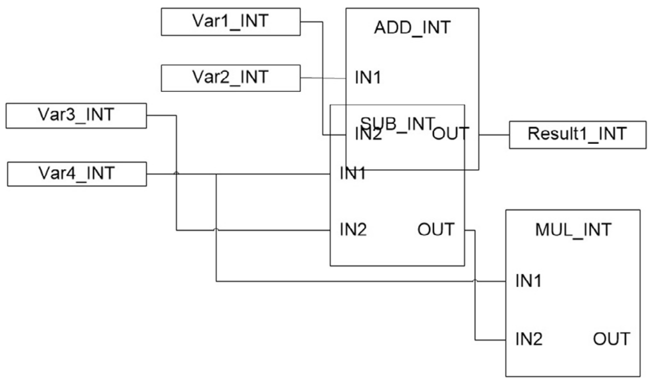


Fig. 13. An example of overlapping diagrams.

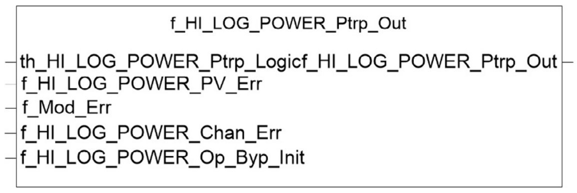


Fig. 14. An example of overlapping port names.

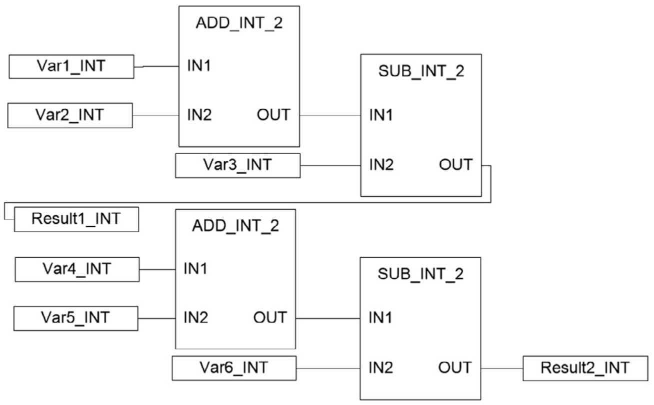


Fig. 15. An example of mislocated output.

able that has a meaningful name, as shown in Fig. 16. The literal 100 on the left-hand side has been replaced by the constant *MAX_-TRIP* variable.

Rule 2.2.5 Feedback variables should be accompanied with additional identifiers

Feedback variables tend to be confused with other types of variables, such as (normal) I/O variables and connections. They need to be distinguished using additional identifiers such as '*FB_*'. Fig. 17 shows an example of a feedback variable with/without identifiers.

Rule 2.2.4 Global variables should be accompanied with additional identifiers

All variables in an FBD are global ones that can access and store a value anywhere in the diagram. On the other hand, a user-defined function block has a strict boundary of local and global variables. We have to distinguish global variables (IEC, 2013) from local ones clearly. Fig. 18 shows examples that employ an identifier '*GB_*' for global variables used in user-defined function blocks *Trip_Calculation* and *Trip_Decision*.

Rule 2.2.6 Connector/continuation should be accompanied by additional identifiers

Connector/continuations express connections between two blocks. They need to be distinguished using additional identifiers such as '*Connection_*' to reduce confusion with other variables, as shown in Fig. 19.

3.3.3. Abstraction

This category consists of two rules that pertain to the number of variables used in accordance with NUREG/CR-6463. It proposes restricting the number of global variables under *MAX_GLOBAL*. It also proposes that the number of input variables of a user-defined function block should be controlled.

2.3 Abstraction

2.3.1 Global variables in a user-defined function block should be less than MAX_GLOBAL

2.3.2 The number of input variables in a user-defined function block should be less than MAX_PARA

Rule 2.3.2 The number of input variables should be less than MAX_PARA

The number of input variables of a user-defined function block can vary according to the design intended by the programmers. However, they should be controlled appropriately from various perspectives. Fig. 20 shows two FBDs that employ user-defined function blocks having 8 inputs and 4 inputs, respectively. However, they exhibit the same behavior.

4. FBD checker

4.1. FBD checker

'*FBD Checker*' is a structural analysis tool for FBD programs that are written in the PLCopen TC6 XML format (PLCopen XML schema Ver. 2.0, 2008). It checks the 56 rules proposed by this paper, as summarized in Table 2, mechanically and PLC platform-independently. It can also explain the explicit correspondence to higher guidelines such as NUREG/CR-6463 and IEC 61131-3.

For example, the '*FBD Checker*' read the FBD in Fig. 21(b), and found 9 violations of 5 sub-categories at 5 locations, as shown in

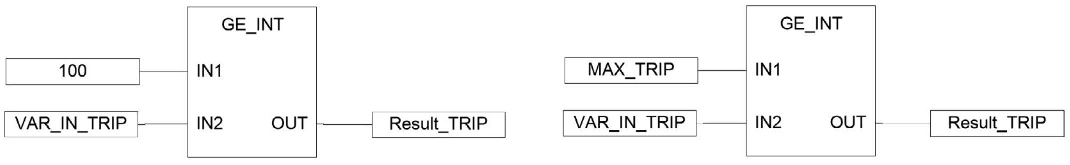


Fig. 16. An example involving the use of a literal as an input variable.

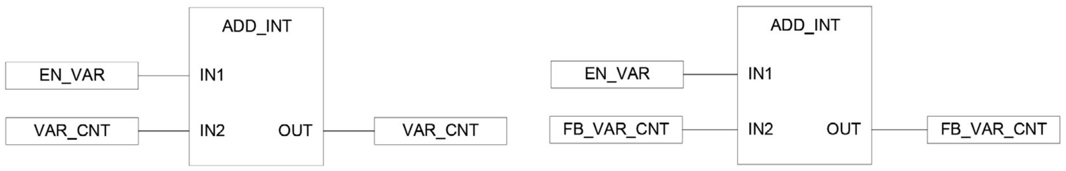


Fig. 17. An example of a feedback variable with/without identifiers.

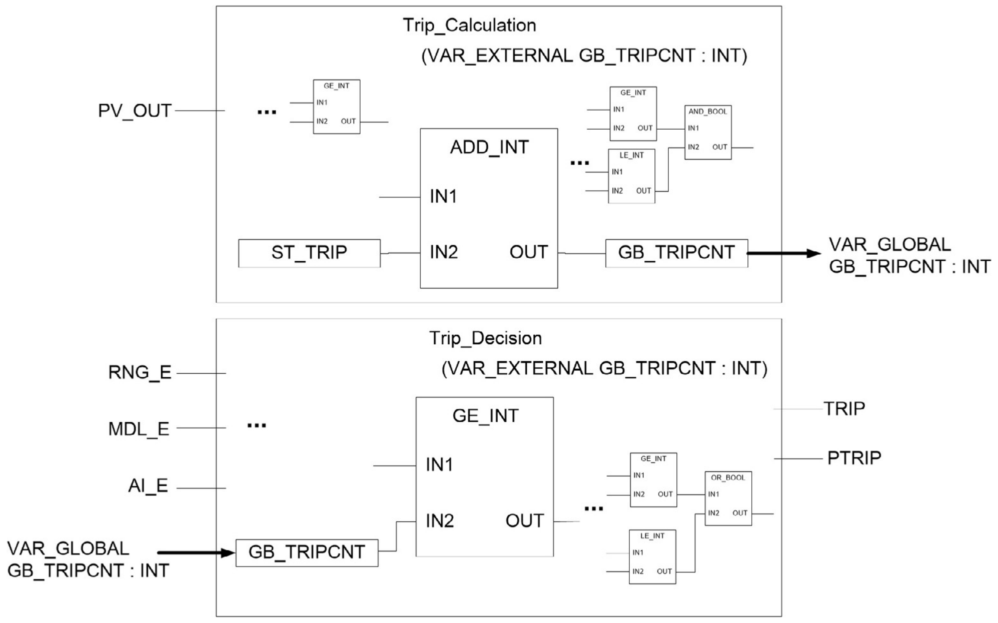


Fig. 18. An example involving the use of an identifier for a global variable.

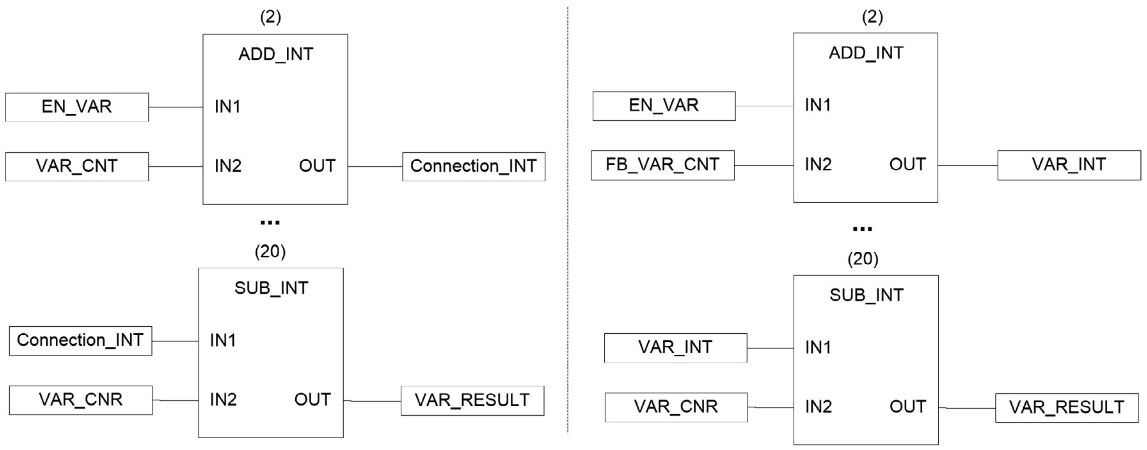


Fig. 19. An example of Connector/Continuation with identifiers.

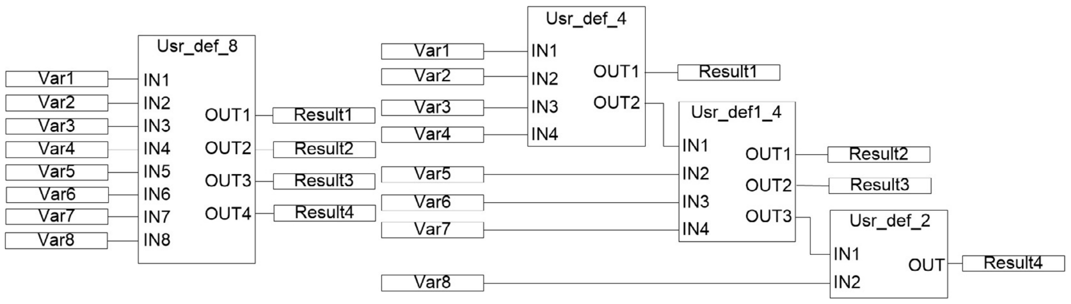
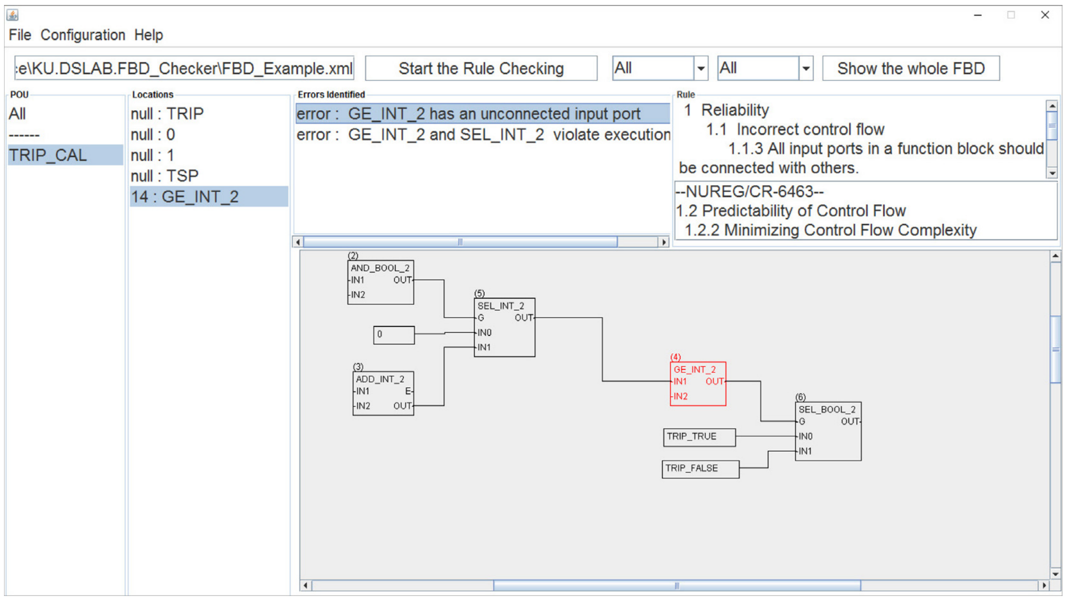


Fig. 20. An example of two different numbers of input parameters.



:e\KU.DSLAB.FBD_Checker\FBD_Example.xml

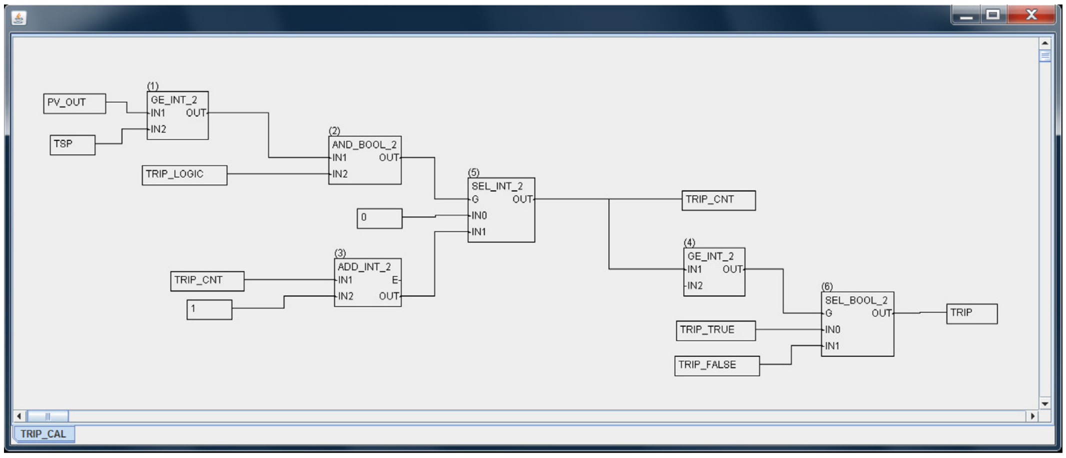
POU	Locations
All	null : TRIP
------	null : 0
TRIP_CAL	null : 1
	null : TSP
	14 : GE_INT_2

error : GE_INT_2 has an unconnected input port
error : GE_INT_2 and SEL_INT_2 violate execution

1 Reliability
1.1 Incorrect control flow
1.1.3 All input ports in a function block should be connected with others.

--NUREG/CR-6463--
1.2 Predictability of Control Flow
1.2.2 Minimizing Control Flow Complexity

(a) A structural checking result of '*FBD Checker*'



(b) The full-sized FBD window

Fig. 21. The '*FBD Checker*'.

Fig. 21(a). As highlighted, `GE_INT_2` has no input variable for the `IN2` port, and the specific rule that was violated (*i.e.*, Rule 1.1.3) is explained. Its corresponding higher guideline "*1.2.2 Minimizing Control Flow Complexity*" can also be acknowledged in NUREG/CR-6463.

Other violations can be understood in a similar way: The execution order of blocks (5) and (4) is not sequential (Rule 1.1.7). Literals such as 0 and 1 are also used directly (Rule 2.2.3). The variable '*TSP*' is too short to satisfy Rule 2.2.1. The '*FBD Checker*' also shows the entire FBD program as well as the part highlighted.

5. Case study

We performed two case studies to demonstrate the effectiveness and validity of the proposed rules and the structural analysis tool, '*FBD Checker*.' They started from a formal requirements specification (KAERI, 2003) and an FBD program (KAERI, 2006) for preliminary versions of KNICS APR-1400 RPS BP in Korea, respectively, but we transformed them into the FBD programs having the PLCopen TC6 XML format. All subsequent artifacts that were generated from the original (KAERI, 2003; KAERI, 2006) have nothing to do the KNICS APR-1400 RPS.

The first case study starts from an *NuSCR* (Yoo et al., 2005) formal software requirement specification (KAERI, 2003. The KNICS project developed the formal requirement specification as well as typical requirement specifications written in natural languages in order to guarantee the system validity through diversity (Yoo and Seong, 2002; Kelly and Murphy, 1990). We used the *NuDE* framework (Kim et al., accepted) to translate it into FBD programs, as depicted in Fig. 22. '*NuSCRtoFBD*' performs the behavior-equivalent translation mechanically, while several formal verification and safety analysis methods are also supported seamlessly. The FBD can be translated into Verilog and C programs for PLC and FPGA platforms, respectively.

On the other hand, the second case study uses one of the official versions of FBDs (KAERI, 2006) for the RPS BP. As the FBD was developed by experts manually from the later requirement specification (Korea Atomic Energy Rearch Institute, 2005), it is closer to the official last release. The FBD was developed in the software engineering environment of its target PLC, *i.e.*, '*pSET*' (Cho et al., 2007) in *POSAFE-Q* PLC of *POSCO ICT* (PONU-Tech, 2015), and we transformed it into the form of the PLCopen TC6 standard with the help of '*pSET2TC6*' (Lee et al., 2011; Jee et al., 2010). Fig. 23 shows the process used of the second case study. The target system consists of 18 independent shutdown logics of nuclear reactor, and two case studies use 5 representative trip logics of nuclear reactors, such as '*fixed set-point rising/falling*', '*variable set-point rising/falling*', and '*manual reset*' trip logics.

5.1. Case study I

Fig. 24 shows an example of the *NuSCR* formal requirements specification in '*NuSRS*' and the translated FBD in '*FBD Editor*' (Kim et al., 2014; Lee et al., 2014). We used the *NuDE* framework to translate the NuSCR specification into equivalent FBD programs. The translated FBD consists of 59 inputs, 43 outputs, and 1,046 function blocks.

Table 4 summarizes the result of the structural checking on the FBD program using the '*FBD Checker*.' It found 4,643 violations for 11 categories (*i.e.*, 4 from the *Reliability* category and 7 from the *Maintainability* category). As the *NuDE* framework did not yet follow the proposed guidelines and rules, it translated the formal specification into an FBD program that contained a number of violations. As explained below, most violations are neither serious nor critical to software safety because they resulted from the immaturity of the non-commercial toolsets of the *NuDE*.

The most frequently violated rules in the *Maintainability* category are (Rule 2.1.6) and (Rule 2.1.7). However, the violations resulted from the immaturity of the '*NuSCRtoFBD*' translator and '*FBD Editor*,' and they may be resolved by improving the maturity of the tools. For example, we could find the violations of Rules 2.1.1, 2.1.6, and 2.1.7 easily at the translated FBD of Fig. 25. Rules such as 2.2.2 and 2.2.6 can also be resolved by improving the tools.

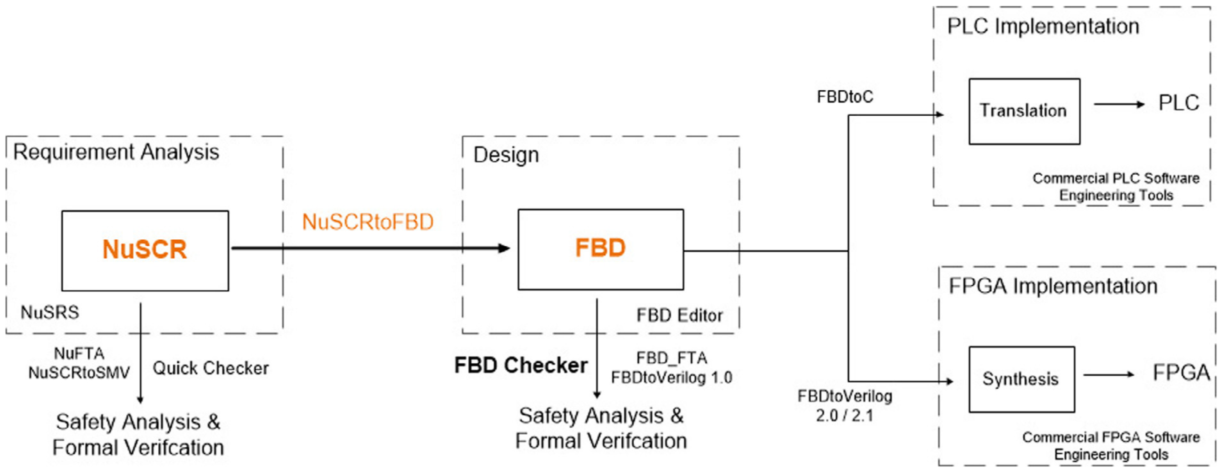


Fig. 22. An overview of case study I in the *NuDE* framework.

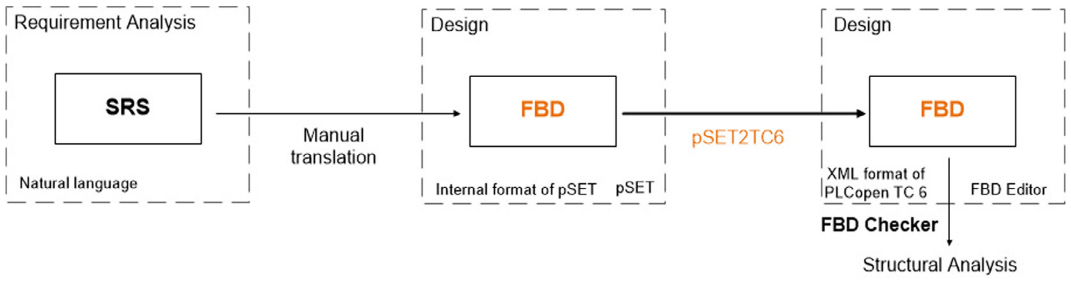
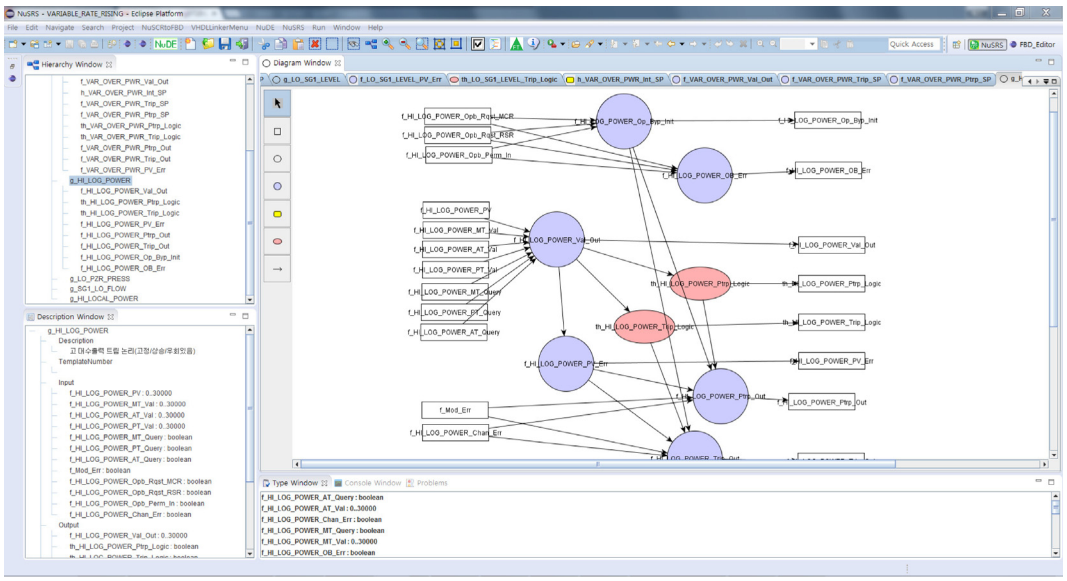


Fig. 23. An overview of case study II.


(a) *NuSRS*

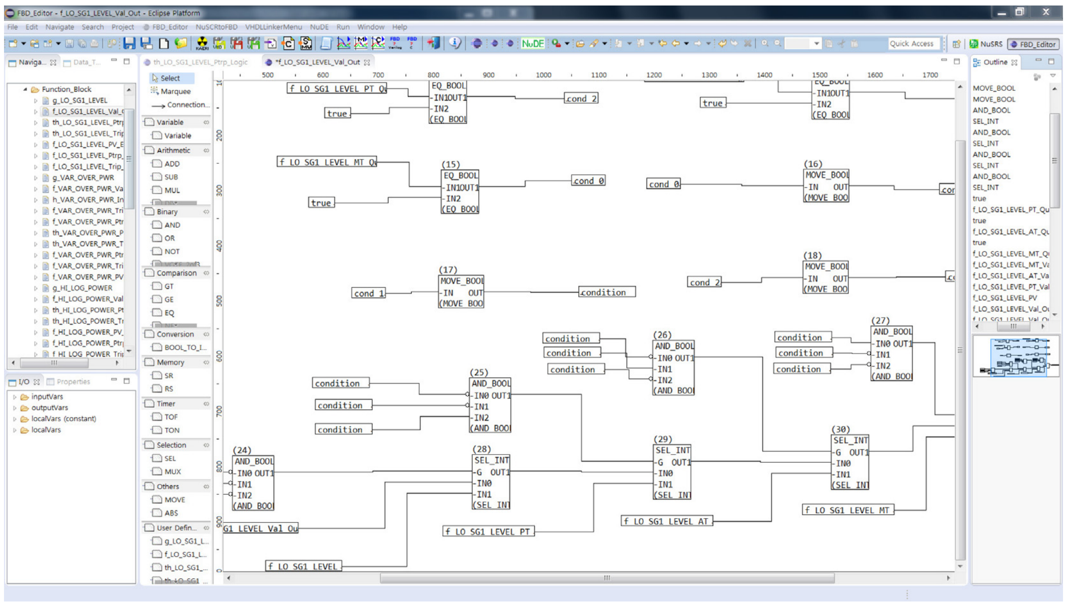

(b) *FBD Editor*

Fig. 24. The NuSCR formal requirements specification and the FBD program translated in the *NuDE*.

Most violations in the *Reliability* category resulted from the "*Implicit-Type Conversion*" (Rule 1.1.4 and 1.3.1). The rules propose to strictly restrict the use of implicit-type conversion, and propose the use of explicit-type conversions for all cases. Because the *NuDE* framework did not yet follow the rules, it produces an FBD containing implicit-type conversions. For example, '*FBD Checker*' produces a violation when an input variable of the derived type 0..30, 000 of (UINT) is connected to an input port of (INT). Other violations such as "*The number of function blocks in an FBD should be less than MAX_FUNCTION*" (Rule 1.1.2) can also be easily resolved in different ways.

In summary, the '*FBD Checker*' found more than 4,000 violations from the FBD program, all of which the *NuDE* framework translated mechanically from an *NuSCR* formal requirements specification for a preliminary version of a Korean nuclear power plant. Most of them resulted from the immaturity of the *NuDE* toolsets that did not consider the guidelines and rules proposed, and were not serious. All violations of the implicit-type conversion were examined to determine whether they were used appropriately according to the intent of programmers, and we found that they were all used correctly.

5.2. Case study II

We also applied the structural analysis of '*FBD Checker*' into the FBD program, which is another preliminary version (KAERI, 2006) of the KNICS APR-1400 RPS BP. Software experts used the software engineering tool, '*pSET*,' for *POSAFE-Q* PLCs to develop the FBD pro-

Table 4
Structural analysis results in case study I.

Reliability	Rules	1.1.2	1.1.4	1.2.3	1.3.1			
	# Violations	3	17	6	512			
Maintainability	Rules	2.1.1	2.1.6	2.1.7	2.2.1	2.2.2	2.2.3	2.2.6
	# Violations	27	620	2482	101	445	1	429

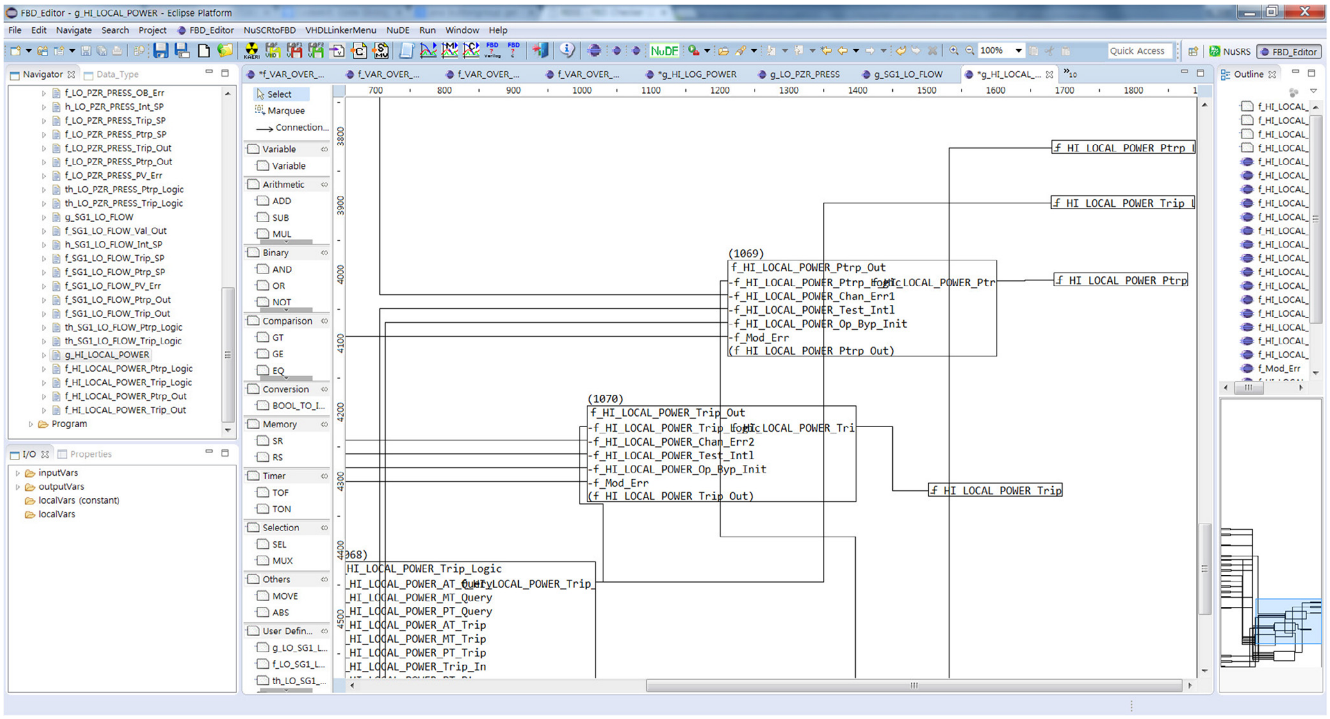


Fig. 25. A screen-dump of the FBD program in '*FBD Editor*,' containing violations of 2.1.1, 2.1.6, and 2.1.7.

gram. We then used '*pSET2TC6*' in the *NuDE* framework to transform it into developed the FBD programs from a revised requirements specification (Korea Atomic Energy Rearch Institute, 2005). We transformed the FBD into a behaviorally equivalent XML form of the PLCopen TC6 with the help of '*pSET2TC6*' in the *NuDE* framework. The case study used 5 representative trip logics used in the former case study, consisting of 119 function blocks. Because the experts developed the FBD from scratch, it is more compact and optimized than the one translated mechanically from formal requirement specifications.

The '*FBD Checker*' produced 87 violations from 4 categories, such as three from the *Reliability* category and one from *Maintainability* category, as summarized in Table 5. In comparison with the former case study, the FBD has smaller, but not simple violations, and most of them can be resolved easily (Rules 1.2.10, 1.2.3, and 2.2.1). However, for the 22 violations of Rule 1.2.1 ("*All local variables should be initialized.*"), there is a need for an explanation about why they did not result in critical errors.

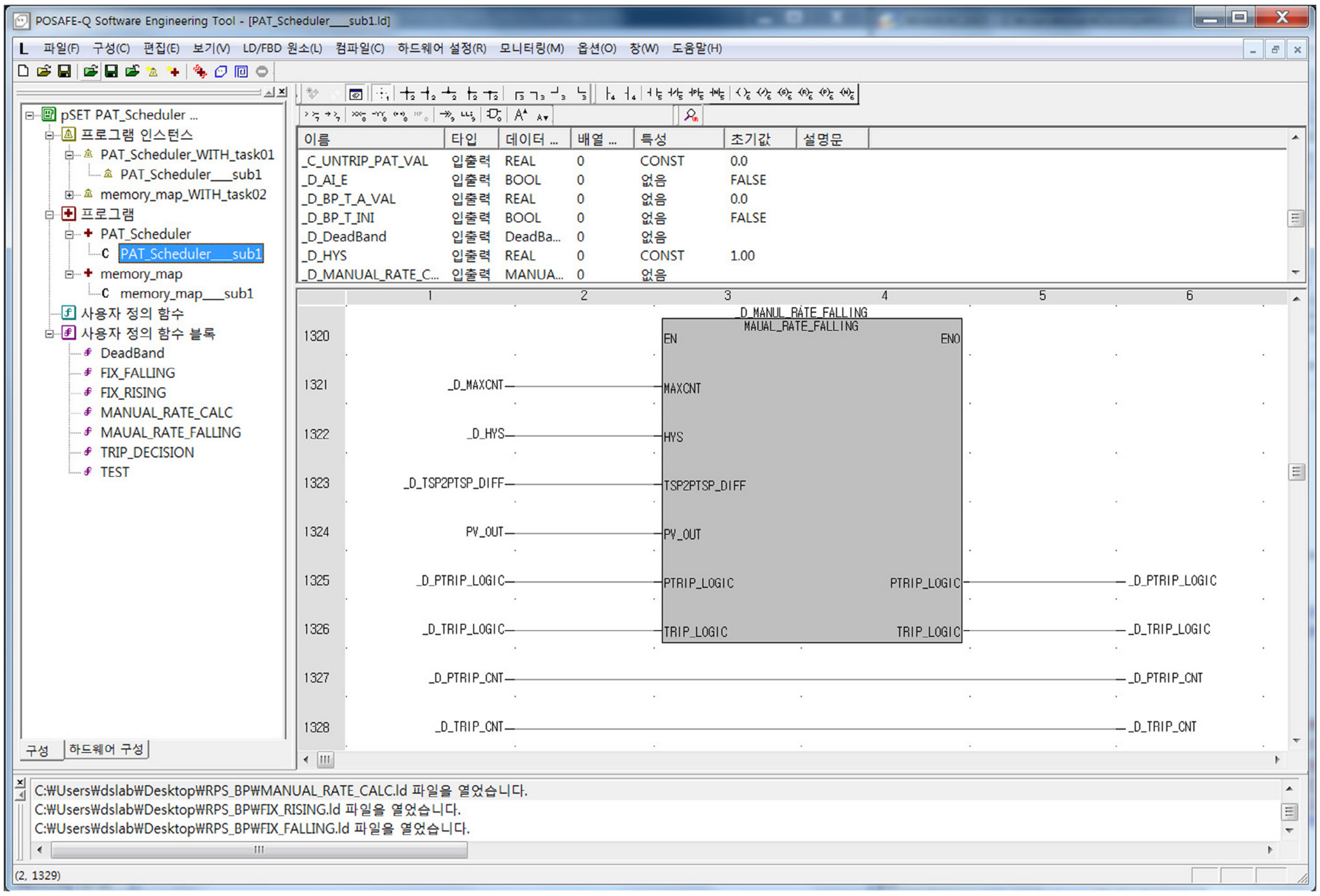
The FBD of case study II was developed with the help of the software engineering tool (*i.e.*, *pSET*) of the target PLC vendor. The tool provides a way to define and initialize all variables in one table, as shown at the top-center window in Fig. 26. Programmers can categorize the variables as *I/O* and *local* types appropriately. While extracting 5 logic parts from the whole FBD of 18 logics, a few of the I/O variables were redefined as local variables. As '*pSET*' does not require I/O variables to be initialized (because they are initialized as NULL in memory), the redefined local variables are also not initialized, resulting in the 22 violations of Rule 1.2.1.

For example, the *_D_HYS* variable is defined as an I/O type in Fig. 26. However, it was redefined as a local variable with no initialization when the user-defined function block MANUAL_RATE_FALLING was extracted from the whole FBD, and it was considered as another stand-alone FBD (*i.e.*, a unit of structural analysis for '*FBD Checker*'). This case resulted in a violation of Rule 1.2.1. On the other hand, the extraction has no effect on the I/O variable *PV_OUT* because its I/O type remains the same after the extraction. The '*FBD Checker*' produced no violation for the case.

In summary, the '*FBD Checker*' found 87 violations from the FBD program, which experts programmed in the '*pSET*' software engineering environment. There were 22 violations of Rule 1.2.1, which resulted from the process of preparing the case study, and 65 other violations could be resolved easily by revising and obeying FBD programming guidelines and rules. It should be noted that the case study produced few violations in comparison with case study I because the PLC software engineering tool provides and follows mature FBD programming guidelines implicitly.

Table 5
Structural analysis results.

Reliability	Rules	1.2.1	1.2.3	1.2.10
	# Violations	22	2	41
Maintainability	Rules	2.2.1		
	# Violations	22		



이름	타입	데이터 ...	배열 ...	특성	초기값	설명문
_C_UNTRIP_PAT_VAL	입출력	REAL	0	CONST	0.0	
_D_AI_E	입출력	BOOL	0	없음	FALSE	
_D_BP_T_A_VAL	입출력	REAL	0	없음	0.0	
_D_BP_T_INI	입출력	BOOL	0	없음	FALSE	
_D_DeadBand	입출력	DeadBa...	0	없음		
_D_HYS	입출력	REAL	0	CONST	1.00	
_D_MANUAL_RATE_C...	입출력	MANUA...	0	없음		

```
C:₩Users₩dslab₩Desktop₩RPS_BP₩MANUAL_RATE_CALC.ld 파일을 열었습니다.
C:₩Users₩dslab₩Desktop₩RPS_BP₩FIX_RISING.ld 파일을 열었습니다.
C:₩Users₩dslab₩Desktop₩RPS_BP₩FIX_FALLING.ld 파일을 열었습니다.
```

Fig. 26. A screen-dump of '*pSET* about variable definitions.

6. Conclusion and future work

In this paper, we propose a set of 56 specific rules on the structure of FBD programs, which can acknowledge the direct correlation with higher guidelines such as NUREG/CR-6463 and IEC 61131-3. The '*FBD Checker*' also performs the structural analysis for any FBD programs in the PLCopen TC6 XML form PLC platform-independently (*i.e., PLC and SW tools*). We used commercial PLC software engineering tools to perform the analysis, but the direct correlations remain unclear. For FBD programs, we also performed case studies of structural analyses that were sampled from preliminary versions of Korean nuclear power plants, and then transformed by the *NuDE* framework. They show the effectiveness and potential of the proposed rule sets and analysis tool. We currently plan to deal with cyber-security (Kim, 2014) guidelines such as US NRC RG 5.71 (NRC, 2010) and IEC 62645 (IEC, 2014; Lee et al., 2013).

Acknowledgements

This research was supported by the Ministry of Science, ICT, & Future Planning. It was also supported 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

Cho, S., Koo, K., You, B., Kim, T., Shim, T., Lee, J., 2007. Development of the loader software for PLC programming. In: Proceedings of conference of the institute of electronics engineers of Korea, vol. 30. (in Korean).
CODESYS, Industrial IEC 61131-3 PLC Programming, www.codesys.com (2015.10).
CODESYS, Static Analysis Data Sheet, http://store.codesys.com/codesys-static-analysis.html?store=en (2015.10)
de Mario, S., 2008. Restricting IEC 61131-3 programming languages for use on high integrity applications. In: Emerging Technologies and Factory Automation (ETFA), IEEE International Conference on, 2008. pp. 361–368.
Functional safety of electrical/electronic/programmable electronic safety-related systems: Part 3. Software requirements (IEC 61508-3), Tech. rep., International Electrotechnical Commission (IEC), (1997).
Invensys, 2006. Developer's Guide TriStation 1131, http://iom.invensys.com/EN/pages/home.aspx.
Invensys, Safety Software Suite TriStation 1131 (TS1131),http://iom.invensys.com (2015.10)
Jee, E., Jeon, S., Cha, S., Koh, K., Yoo, J., Park, G., et al., 2010. FBDVerifier: interactive and visual analysis of countereample in formal verification of functino block diagram. J. Res. Pract. Inf. Technol. 42 (3), 255–272.
Jung, S., Lee, D.-A., Kim, E.-S., Yoo, J., Lee, J.-S., 2014. Programming Guidelines for FBD Programs in Reactor Protection System Software. In: Transactions of the Korean Nuclear Society Autumn Meeting, Pyeongchang, Korea. pp. 1986–1988.
KAERI, 2003. Software Design Specification for Reactor Protection System KNICS-RPS-SRS101, Tech. rep., Korea Atomic Energy Research Institute, rev.00.
KAERI, 2006. Software Design Specification for Reactor Protection System KNICS-RPS-SD231-01, Tech. rep., Korea Atomic Energy Research Institute, rev.02.
Kelly, J.P.J., Murphy, S.C., 1990. Achieving dependability throughout the development process: a distributed software experiment. IEEE Trans. Software Eng. 16 (2), 153–165.
Kim, D.-Y., 2014. Cyber security issues imposed on nuclear power plants. Ann. Nucl. Energy 65, 141–143.
Kim, J., Lee, D.-A., Seo, Y.-J., Yoo, J., 2014. NuSCRtoFBD 4.0: Automatic Generation of FBD Program for FPGA development from NuSCR Formal Specification. In: Korea Computer Congress. pp. 1986–1988 (in Korean).
Kim, E.-S., Lee, D.-A., Jung, S., Yoo, J., Choi, J.-G., Lee, J.-S. NuDE 2.0: A formal-methods based software development, verification and safety analysis environment for digital I&Cs. J. Comput. Sci. Eng. (accepted).
Korea Atomic Energy Rearch Institute, 2005. SRS for Reactor Protection System, KNICS-RPS-SRS221 Rev. 00, in Korean.
Lee, D.-A., Yoo, J., 2011. pSET2TC6: A Translater Tool to Standardize the Output Format of pSET. In: The Korean Intitute of Information Scientists and Engineers, vol. 38. pp. 105–107 (in Korean).
Lee, J.-H., Kim, E.-S., Yoo, J., Lee, J.-S., 2013. A Preliminary Report on Static Analysis of C Code for Nuclear Reactor Protectino System. In: IFAC Conference on Manufacturing Modelling, Management, and Control, 2013, pp. 2193–2198.

Lee, D.-A., Yoo, J., Lee, J.-S., 2014. Guidelines for the Use of Function Block Diagram in Reactor Protection Systems. In: The 21st Asia-Pacific Software Engineering Conference (APSEC). pp. 141–149.

Lee, D.-A., Kim, E.-S., Yoo, J., Lee, J.-S., 2014. FBDEditor: an FBD design program for developing nuclear digital I&C systems. In: Korea Conference on Software Engineering. pp. 315–318 (in Korean).

Nuclear power plants – Instrumentation and control systems – Requirements for security programmes for computer-based systems(IEC62645), Tech. rep., International Electrotechnical Commission (IEC), 2014.

NUREG/CR-6463: Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems, Tech. rep., United States Nuclear Regulatory Commission (NRC), rev. 1, 1997.

NUREG/CR-6463: Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems, Tech. rep., United States Nuclear Regulatory Commission (NRC), 1997.

NRC, United States Nuclear Regulatory Commission, http://www.nrc.gov (2015.10)

Part 1: Concepts and Function Blocks, Tech. rep., PLCopen TC5: Safety Software Technical Specification, 2006.

Part 2: User Examples, Tech. rep., PLCopen TC5: Safety Software Technical Specification, 2008.

PLCopen XML schema Ver. 2.0, 2008. Tech. rep., PLCopen TC6: XML work.

PONU-Tech, Nuclear Plant Design and Repair Services, http://www.ponu-tech.co.kr/ (2015.10)

Prahofer, H., Angerer, F., Ramler, R., Lacheiner, H., Grillenberger, F., 2012. Opportunities and challenges of static code analysis of IEC 61131-3 programs. In: Emerging Technologies and Factory Automation (ETFA), IEEE International Conference on, 2012, pp. 1–8.

Programmable Controllers: Part 3. Programming Languages (IEC 61131-3), Tech. rep., International Electrotechnical Commission (IEC), 2013.

Programmable Controllers: Part 8. Guidelines for the application and implementation of programming languages (IEC 61131-8), Tech. rep., International Electrotechnical Commission (IEC), 1993.

Regulatory Guide 5.71: Cyber Security Programs for Nuclear Facilities, Tech. rep., United States Nuclear Regulatory Commission (NRC), 2010.

Safety I&C system platform in the nuclear power plant, AVERA, www.de.avera.com (2015.8).

SIEMENS, PLC Control Systems, http://www.w3.siemens.com (2015.10)

SIEMENS PLC Control Systems, SIMATIC-Manager Programming with STEP7 manual, www.siemens.com (2015.10).

Yoo, C.S., Seong, P.H., 2002. Experimental analysis of specification language diversity impact on NPP software diversity. J. Syst. Softw. 62 (2), 111–122.

Yoo, J., Kim, T., Cha, S., Lee, J.-S., Son, H.S., 2005. A formal software requirements specification method for digital nuclear plants protection systems. J. Syst. Software 74 (1), 73–83.

Yoo, J., Kim, E.-S., Lee, D.-A., Choi, J.-G., 2014. An Integrated Software Development Framework for PLC & FPGA based Digital I&Cs. In: International Symposium on Future I&C for Nuclear Power Plants/International Symposium on Symbiotic Nuclear Power System (ISOFIC/ISSNP).

Yoo, J., Kim, E.-S., Lee, D.-A., Choi, J.-G., Lee, Y.J., Lee, J.-S., 2014. NuDE 2.0: A Model-based Software Development Environment for the PLC & FPGA based Digital Systems in Nuclear Power Plants. In: International Symposium on Integrated Circuit. pp. 604–607.