

Software safety analysis of function block diagrams using fault trees

Younju Oh^a, Junbeom Yoo^{b,*}, Sungdeok Cha^b, Han Seong Son^c

^aLG Electronics Inc., LG R&D Complex, 553, Hoggae-1dong, Dongan-gu, Anyang-shi, Kyongki-do, Republic of Korea

^bDepartment of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST) and AITrc/SPIC/IIRTRC, 373-1, Gusong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea

^cKorea Atomic Energy Research Institute (KAERI), I&C and HF team, 150, Deokjin-dong, Yusong-gu, Taejeon, Republic of Korea

Received 4 February 2004; accepted 28 July 2004

Available online 19 October 2004

Abstract

As programmable logic controllers (PLCs) are often used to implement safety-critical embedded software, safety demonstration of PLC code is needed. In this paper, we propose a fault tree analysis technique on Function Block Diagrams (FBDs) which is one of the most widely used PLC programming languages. FBD is currently being used to develop Reactor Protection System (RPS) for a nuclear power plant in South Korea. Our approach to fault tree analysis, which combines fault-oriented and cause/effect-oriented viewpoints, is easy to understand and offers systematic guidelines to ensure safety of PLC code. Domain experts found the approach to be useful through a case study on RPS, and this paper compares completeness and comprehensiveness of the semi-automatically generated fault trees using the proposed approach against the one manually prepared by nuclear safety engineers.

© 2004 Elsevier Ltd. All rights reserved.

Keywords: Safety analysis; Fault tree analysis; Function block diagram; Programmable logic controller

1. Introduction

Software is increasingly being used to handle safety-critical system functions that were previously controlled by humans or hardware in the past. As a large number of hazards in such systems are known to be caused by software that controls it, safety analysis is often required on safety-critical embedded software [1]. In this paper, we focus on software safety analysis of Function Block Diagram (FBD) [2] programs. FBD is a standard application programming language for the Programmable Logic Controller (PLC) [3]. PLC is currently being used in the development of Reactor Protection System (RPS) by KNICS [4] in Republic of Korea.

RPS software development starts by first analyzing preliminary system requirements written in natural language and performing safety analysis techniques such as FMEA [5] and HAZOP [6] on the requirements. Software requirements are then converted to FBD language which

PLC tools can interpret and compile. Software HAZOP and software Fault Tree Analysis (FTA) [7] on the FBD design are usually performed before executable code is automatically generated. Manual application of FTA on FBDs is not only labor-intensive but also potentially error-prone because quality of analysis depends heavily on the capability of analyst. In addition, primitive notation of FBD makes the exhaustive identification of faults that can occur in FBDs difficult.

To analyze FBD programs efficiently, we define fault tree templates for each FBD function block and propose a semi-automatic FTA process. The templates combine two different views, *fault-oriented view* and *cause/effect (CE)-oriented view*. In FBDs, faults may occur because incorrect FBD blocks are used or input and output paths are incorrectly connected. Incorrect intermediate values are propagated through the FBD network to the final output and an undesired event may occur. The fault-oriented view shows feasible faults in FBD design. The CE-oriented view illustrates how incorrect values may be propagated through FBD network.

The proposed approach makes several assumptions to control complexity of fault trees and focuses on faults that

* Corresponding author.

E-mail address: jbyoo@salmosa.kaist.ac.kr (J. Yoo).

are likely to occur. That is, definition of each function block is assumed to be correct. For example, AND function block is identical to the AND gate in the logic design, and we do not address faults that may be caused by errors in implementing AND function block. In the RPS design written in FBDs, only a small number of well-defined FBD blocks are used, and they have been extensively used in industry. Therefore, we focus more on the possibilities of inputs incorrectly connected or incorrect FBD blocks used (e.g. using AND where OR is required). In addition, our approach focuses on logical correctness of the FBD design and do not address hazards that may be caused by common mode failures or hardware failures. While such failures are feasible in theory, nuclear applications provide adequate protection against such failures through fault-tolerant design.

The proposed approach was applied to the safety analysis on partial design of RPS. Faults leading to the undesired system states, which were often omitted in manual safety analysis, could be identified. Our fault tree was semi-automatically generated, and domain experts found it to be easy to understand and review.

The paper is organized as follows. In Section 2, we briefly review the Function Block Diagram and related work on software fault tree analysis. In Section 3, we categorize all feasible faults in FBDs based on our experience with FBD design and analysis. Section 4 illustrates how two different and complementary viewpoints to fault tree analysis can be effectively combined into template definitions. Section 5 describes fault tree analysis procedure using the proposed approach and case study which explains how our approach was used on RPS design through a comparison between the proposed approach and manually developed fault trees. Finally, in Section 6, primary

contributions of this research are summarized and potential research topics are discussed.

2. Related work

2.1. Function block diagrams

Programmable Logic Controllers (PLC) [3] are widely used in diverse control systems in chemical processing plants, nuclear power plants or traffic control systems. A PLC, an industrial computer specialized for real-time applications, is an integrated system containing a processor, main memory, input modules and output modules that are coupled together by a common bus.

There are several PLC programming languages. The IEC 61131-3 [2] standards include five: Structured Text (ST), Function Block Diagram (FBD), Ladder Diagram (LD), Instruction List (IL) and Sequential Function Chart (SFC). The FBD is one of the most widely used languages because of its graphical notations and usefulness in applications involving a high degree of information or data flow between control components, that can be designed as a network of software blocks.

FBD design expresses system behavior in terms of flow of signals. Functions between input variables and output variables are graphically represented by a collection of function blocks ‘wired’ together in a manner resembling a circuit diagram. A function block is depicted as a rectangle and input/output variables are connected. Function blocks are classified into categories according to the operations they perform and several types of blocks exist in a category. Fig. 1 shows some of the groups of function blocks and example blocks in each group. The RPS being developed at KNICS [4] is programmed with only the five categories

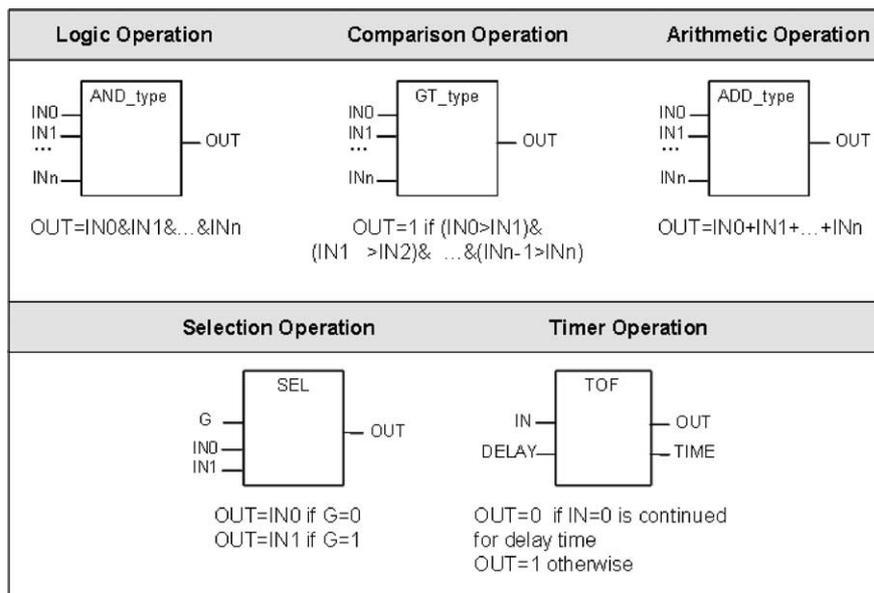


Fig. 1. Categorized example of FBD function blocks.

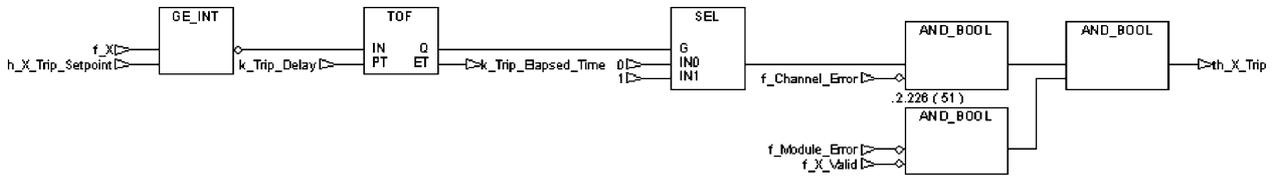


Fig. 2. FBD example.

shown in Fig. 1. Using just these groups increases readability and understandability which therefore increases the software safety in nuclear domains.

Fig. 2 shows a network of function blocks. The output th_X_Trip is produced by the combination of the function block operations. First, the GE function compares inputs f_X and $h_X_Trip_Setpoint$. The result is inverted and given as an input to the TOF function. Next, the TOF outputs a result with the input from the preceding GE block and the delay time k_Trip_Delay based on its function in Fig. 1. The output is given to the following SEL function block. The SEL outputs a 0 or 1 based on the output from the TOF function. Finally, the result from the SEL function and the inverted values of $f_Channel_Error$, f_Module_Error and f_X_Valid are logically AND-ed. The AND-ed result is given to the final output variable th_X_Trip .

2.2. Software fault tree analysis

Fault tree analysis on hardware design is a mature topic [8] used Digraph as an intermediate model to capture the flow of design errors and proposed a fault tree analysis method for Gas Regulation System (GRS). Because Digraph method could handle only static errors, Kocza and Bossche [9] extended the technique to handle dynamic behavior of system using Operating Vector (OV) on static fault trees. In system engineering research, Apostolakis et al. [10] proposed a modeling and analysis method for software controlled embedded systems using Dynamic Flowgraph Methodology (DFM). Developed to specify and analyze industrial process, this method was applied on Titan II Space Launch Vehicle Digital Flight Control System [11] and later extended to support dependability analysis [12] and hazard analysis [13].

Research on software fault tree analysis, unfortunately, is not as mature as the hardware counterpart. [14–16] extracted software fault tree template definitions for various Ada programming language constructs. The templates, equivalent to failure semantics of the Ada statements, offer analyst suggestions on how various statements might cause or contribute to the failure. It provides a semi-automatic approach that relieves the extra effort required in manual fault tree generation.

Papadopoulos proposed Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) that constructs fault trees from an architectural diagram similar to data-flow diagram [17]. However, it lacks details at the architecture and component levels. He also extended

HiP-HOPS to handle Matlab–Simulink models [18]. Sullivan [19] proposed a fault tree construction method from an architecture description language called Reliability Imbedded Design Language (RIDL) and developed an analysis tool, called Galileo [20]. It analyzes module and component information only.

3. Faults in function block diagrams

In FBDs, inputs are passed through various function blocks and the combinations of the block operations result in an output. Therefore, to completely analyze FBDs and improve software safety, one must investigate all safety factors and identify the possible faults in FBDs related to the safety factors. Utilizing software failure mode taxonomy work by Li et al. [21], we defined and categorized the possible faults in FBDs. They are defined based on the characteristics of each function block and the opinions of FBD design experts on where errors occur most frequently.

3.1. Possible faults in function block diagrams

Faults can occur from a combination of function blocks, or from a single function block. Possible faults in FBDs that are defined in this work are as follows:

(1) *Input or output faults*. This fault is similar to the input/output failure modes defined in software failure mode taxonomy work by Li et al. [21]. Detailed possibilities include incorrect input/output variables or values, switched inputs, untimely inputs/outputs and extreme inputs. First, *incorrect input/output variables or values* addresses the cases that a wrong variable or value is given as an input to a function block. If the initial value assigned to variable is incorrect, it is also considered an input fault. Second, *switched inputs* means that ordering of inputs to comparison, selection, or arithmetic blocks must be correct. If inputs IN0 and IN1 to LT (less than) block were accidentally *switched*, the output would be obviously incorrect. Third, incorrect timing of inputs or outputs may cause faults. Errors also occur when extreme (e.g. maximum or minimum possible) values are given as inputs. If two variables are compared, there would be no point in comparing if one was accidentally given an extreme value because the output will always be the same regardless of the value of the other input.

(2) *Incorrect operation or comparison*. Errors can occur when the wrong arithmetic operation or incorrect

comparison is performed. In particular, inclusion/exclusion of ‘=’ is often the most frequent error found in manual FBD design. This type of fault is similar to failure modes such as ‘Incorrect realization of one of the attributes in a function’ and ‘Introduction of an attribute not specified in the requirements’ covered in [21].

(3) *Omission/misplacement.* Omission or misplacement faults are frequently made mistakes by FBD designers. Inverters are especially prone to errors. Another possibility includes omission of an input included in the previous phase (e.g. software requirements specification) but accidentally left out in the software design phase and therefore missing in the FBD design. When compared to research reported in [21], this fault corresponds to ‘Omission of one of the attributes for a function’ in ‘Omission of a function’ failure mode.

(4) *Routine errors.* These faults correspond to ‘Inter-Interaction among functions’ failure modes and ‘Multiple interactions’ failure modes of [21]. A set of connected blocks, which we call FBD networks or routines, have dependency among them. If inputs to a particular FBD block are incorrect, possibilities include propagation of an incorrect output value from connected FBD blocks or ‘stuck-at’ fault.

(5) *Other errors.* Possibilities include runtime errors, internal errors of function blocks, range error (input is outside the range of the data type) and type error (wrong type of input). In [21], runtime errors are captured as either ‘Failure modes due to resource competition’ or ‘platform physical failure modes’. Internal errors of function blocks are included in the ‘Incorrect realization of a function’ failure mode. Range and type errors are covered in ‘Input/Output failure’ failure modes.

It should be noted that the faults defined in this work cover almost all the failure modes included in [21] except

‘Environmental impact factors’. The factors are beyond the scope of this work because FBD-based fault trees deal with only the faults in software design.

3.2. Categorization of faults in function block diagrams

As exhaustive categorization of all possible faults that may occur in FBD is impractical, we make some assumptions in the fault tree analysis. For example, as most commercial FBD design tools provide checks on input types and missing links in the FBD design, we do not consider these errors in the proposed technique. We also do not consider internal errors of function blocks because such likelihood, while theoretically possible, is remote enough that such risk is acceptable. Fig. 3 categorizes the faults mentioned in Section 3.1 and shows which are ‘not considered’, ‘dependent on function blocks’ and ‘independent of function blocks’.

The columns of Fig. 4 show five different FBD groups and the shaded boxes illustrate feasible faults. If specific faults may occur only on input values of certain type (e.g. inverter error on logical or comparison blocks), such constraints are explicitly noted. Similarly, errors involving switched inputs may occur among the comparison blocks only if correct ordering of inputs is critical as it is the case with FBD blocks such as GT, GE, etc. When comparing equality or inequality, switched input faults do not matter. Fig. 4 explicitly notes that such possibilities do not exist by not shading the corresponding cell. Similarly, faults involving extreme inputs are applicable only on FBD blocks belonging to comparison or arithmetic groups. Careful analysis of FBD blocks, taking the characteristics of operations into consideration, and feasible failure modes are captured in the combined fault tree template definitions as explained in Section 4.

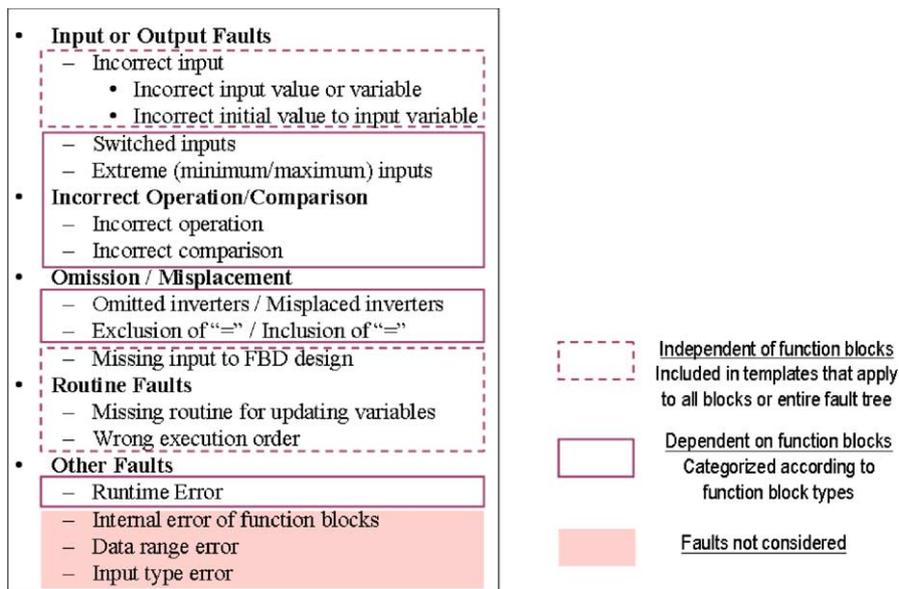


Fig. 3. Categorization of possible faults in FBD.

FBD Blocks \ FBD Faults		Logic		Comparison		Arithmetic		Selection	Timer
		GE,GT,LT,LE	EQ,NE	ADD, MUL	SUB/DIV,MOD				
Runtime Error	Runtime Error					If output is out of range	If output is out of range / If IN1=0		
Input Error	Switched Inputs								
	Extreme Inputs								
Inverter Error	Omitted/Misplaced INV	If block type is boolean	If block type is boolean	If block type is boolean					
Operator/Comparator Error	Incorrect operator / "="								
	Incorrect comparator								

Fig. 4. Possible FBD faults for various FBD blocks.

4. Combined template-based fault tree for function block diagrams

4.1. Combined viewpoints for FBDs

Templates defined for each FBD block include fault-oriented as well as cause/effect-oriented viewpoints. The *fault-oriented view* includes feasible or likely FBD faults. *CE-oriented view* is derived from functional definition, so it is effective in visually illustrating how faults might be propagated. They overlap in some aspects but help analysts perform more comprehensive safety analysis. Templates serve roles similar to that of checklists used in inspection. Combined templates are illustrated using the SEL (select) block whose definition was shown earlier in Fig. 1. Fig. 5(a) is the fault-oriented view template where the root node is the failure of the SEL output, ‘incorrect output of SEL’. The grey nodes ‘incorrect input’, ‘switched inputs’ and ‘omitted inverters’ are the possible faults that can lead to the result, making ‘G=0’ and ‘IN0’ the causes of ‘output is IN0’. In other words, Fig. 5(b) represent the function and/or the fault propagation path of the SEL block.

The template containing both views for the SEL block is shown in Fig. 6, and there are three types of nodes to visually show the two views:

- cause nodes (white): causes of upper event
- fault nodes (grey): faults that can cause undesired event
- conditional fault nodes (dotted grey): faults that occur under certain conditions. For example, fault due to missing INV is applicable only when the input type is Boolean and the inverter does not already exist in the FBD node being analyzed.

Fig. 6 shows the combined SEL template. The causes of the SEL output are the same as Fig. 5(b). The possible faults are also the same as Fig. 5(a). However, ‘incorrect input’ is not included in this template because it is not a fault specific to the SEL function. This is included in another template which will be described later on.

4.2. Combined templates for FBDs

When defining templates corresponding to basic FBD blocks, we make a couple of assumptions so that compact fault tree may be generated. Assumptions made

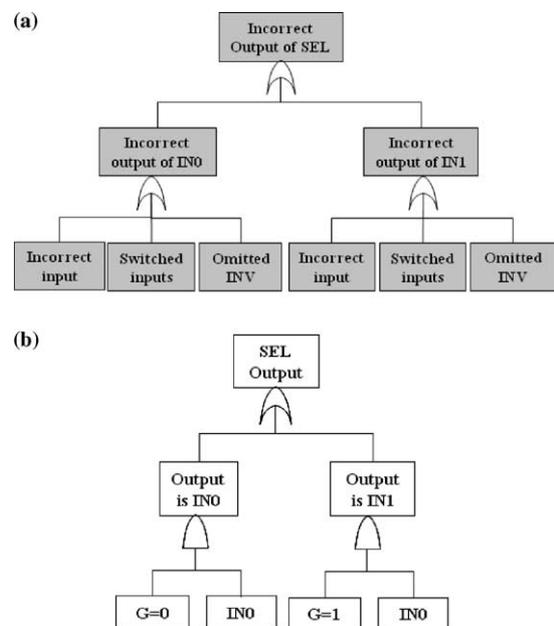


Fig. 5. Two complementary fault tree views: (a) Fault-oriented view and (b) CE-oriented view.

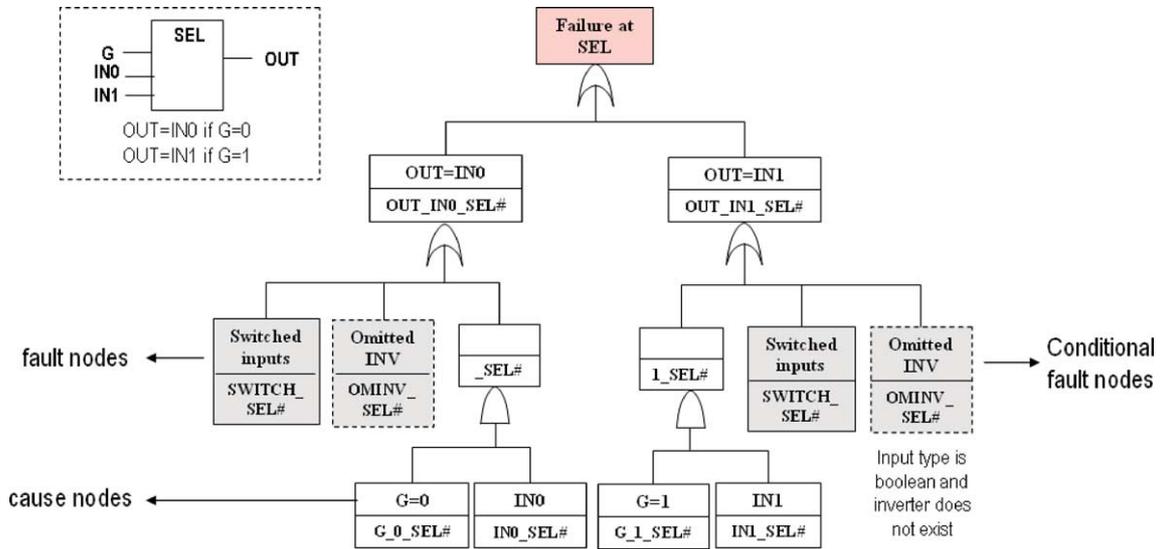


Fig. 6. Combined SEL template.

in the template definitions are realistic and do not compromise quality of safety analysis:

First, function blocks are assumed to have the correct number and type of inputs. FBD tools provide capability to detect such errors as well as the range checks. In addition, we assume that each FBD block correctly performs the required operation. Second, an inverter (INV) placed on the output channel of a FBD block is treated as an input to the FBD block connected by output–input dependency relations. Such interpretation does not change the correctness of fault tree but simplify the definition of fault tree templates. Furthermore, we assume that there exists only one inverter on a connection line because two consecutive inverters cancel each other out in semantics. As an inverter might be missing in the final output block, such case is addressed separately.

The constructed templates cover the five function block groups plus a few others that have been used in

the development of the RPS. Template nodes are named and the label is written on the bottom of each node to improve understandability of fault tree. The following templates are also defined in addition to the ones for FBD blocks:

- *Top event*: to include faults that apply to the entire FBD specification.
- *Each function block*: to include faults specific to a function block. Faults in the categorization table are included here.
- *Terminal node*: to include faults that apply to all function blocks.

(1) *Top event template*. Faults that apply to the entire FBD design, *missing input* and *omitted INV at output* are included here. The *omitted INV at output* node is attached only when the final output block does not have an inverter on its output line and the block type is Boolean. It is

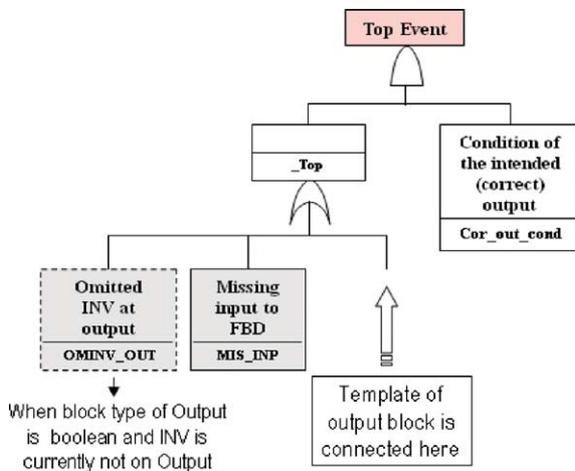
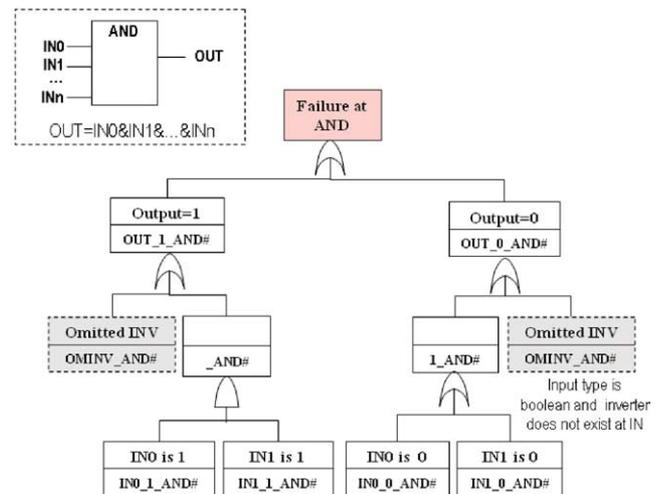


Fig. 7. Top event template.



* Block type boolean

Fig. 8. Logic AND template.

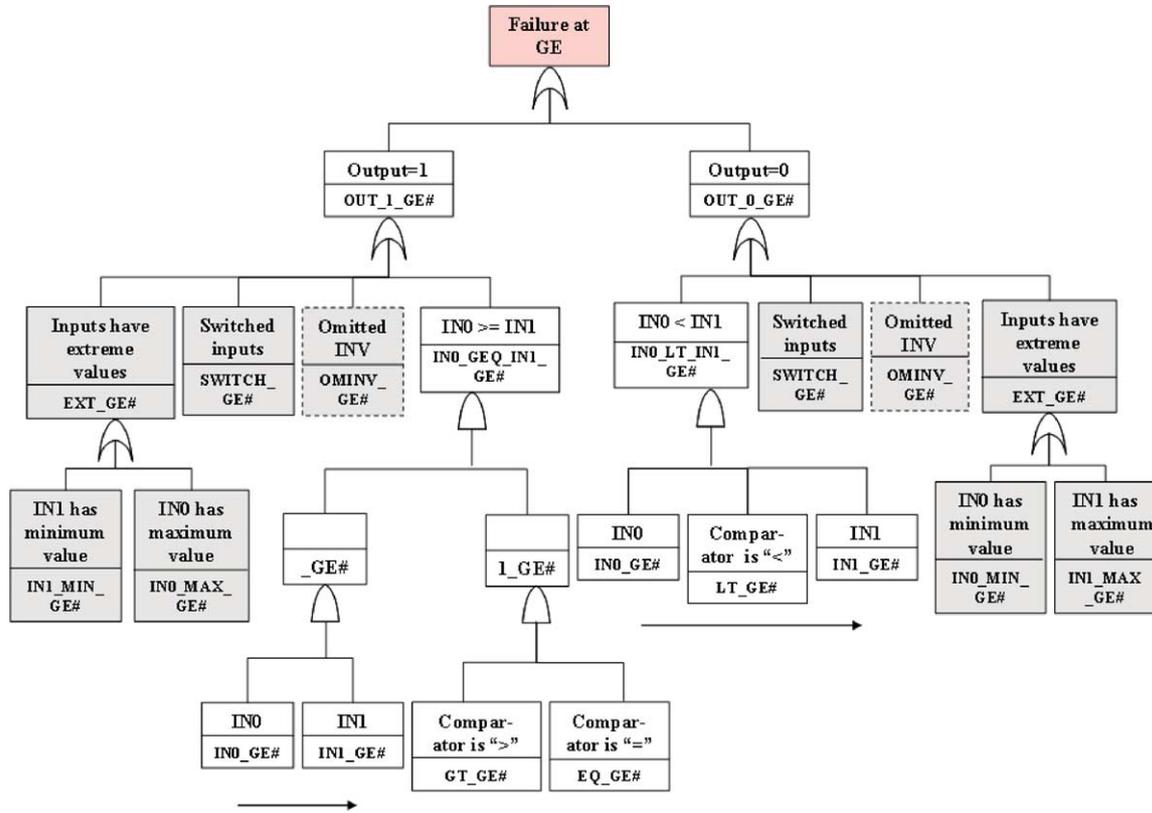


Fig. 9. Comparison GE template.

AND-ed at the root node, to show that the current design is not the correct design for the intended output (Fig. 7).

(2) *Logic block templates.* Logic blocks include AND, OR, NOT, and XOR. As logical blocks produce a Boolean output, only the fault *omitted INV* is included in these templates. When more than two inputs exist for the AND or OR blocks, leaf nodes can be added to the templates. Fig. 8 shows the template of the AND logic block.

(3) *Comparison block templates.* Comparison blocks include GE (greater than or equal), GT (greater than), LE, LT, EQ (equal) and NE (not equal) functions. Our templates consider the blocks to have two inputs. The arrow below the cause nodes are adapted to show that the nodes in the fault tree should always be in that order. For instance, in Fig. 9, ‘INO=value’ and ‘IN1=value’, ‘Comparator is >’ and ‘Comparator is =’ in order means ‘INO >=IN1’. The *switched inputs* fault is not included in the EQ and NE template because the order of the inputs does not affect the output of the EQ and NE block. Fault tree template for GE block appears complex because there are a variety of ways in which the GE block may contribute in generation of an incorrect output although the operation itself is extremely simple. The fact that a seemingly simple FBD block can have complex failure scenarios convincingly illustrates the benefits of having them explicitly defined.

(4) *Arithmetic block templates.* Arithmetic blocks include ADD, MUL, SUB, DIV and MOD. Runtime errors are included in the arithmetic templates. The SUB function

is shown in Fig. 10. When more than two inputs exist for the ADD and MUL blocks, leaf nodes can be added to the templates. The *switched inputs* fault is not included in the ADD and MUL templates as such faults do not generate incorrect output.

(5) *Selection block templates.* The binary selection (SEL) function transfers IN0 to the output when $G=0$ and IN1 when $G=1$, and the multiplexer (MUX) function transfers IN n to the output when $K=n$. The template of SEL is in Fig. 6. As the SEL function is often used to explicitly show the output value is 1 or 0, we constructed a separate function

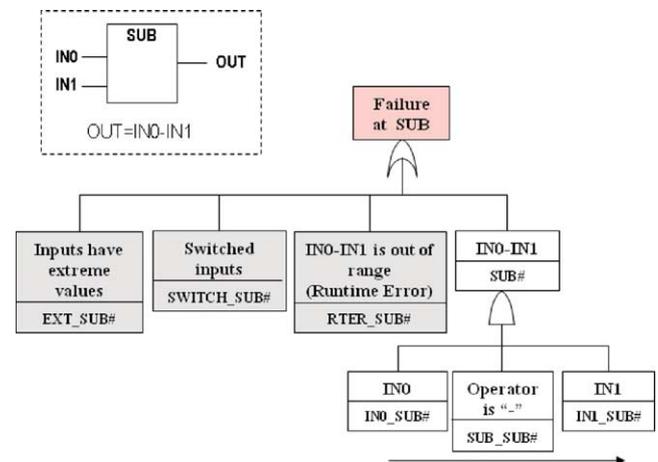


Fig. 10. Arithmetic SUB template.

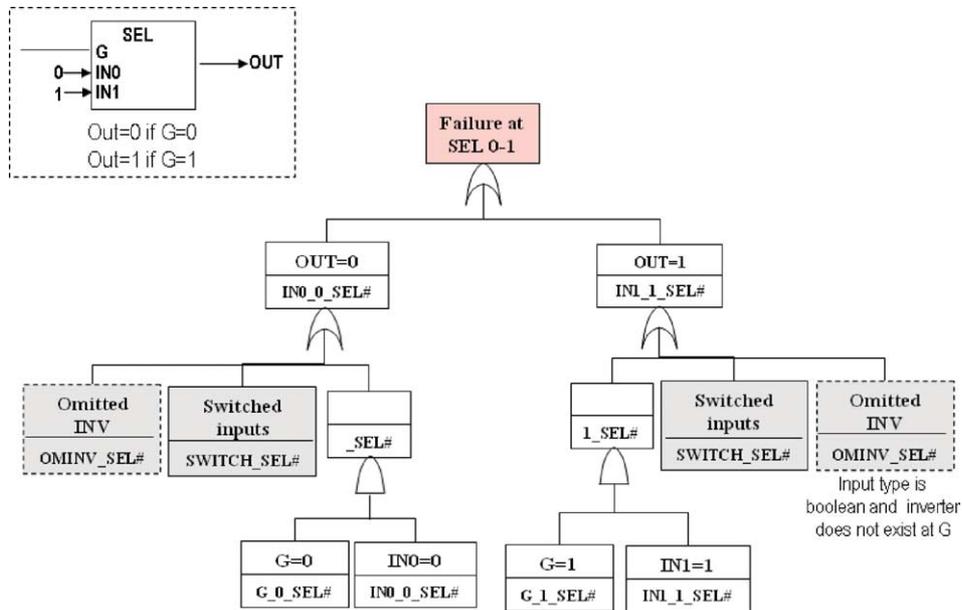


Fig. 11. SEL 0-1 template.

block template in Fig. 11 which shows that if G is 0 then output is 0 and output is 1 when G is 1. The MUX template is similar to the SEL template except that it has n number of subtrees and not just two.

(6) *Timer block template.* The TOF function outputs a 0 when $IN=0$ is continued for delay time (PT) and 1 otherwise. If the changes of all variables throughout time were considered, the fault tree would become too complex to make analysis meaningful and reviewable. Therefore, we consider only the changes of the variable to the input IN of the timer block as time passes. Possibilities, as shown in

Fig. 12, are ‘failed to last for some time period’ and ‘incorrect delay time (incorrect value of PT)’.

(7) *Terminal node templates.* Faults that are not specific to FBD blocks but applicable to all blocks are included in this group. The ‘variable/value’ template has *routine faults* and *incorrect input variable or value* faults. The subtree with the conditional fault node, *incorrect value of input variable*, is added when the top node of the terminal template is a variable node and not a value node. The ‘comparator/operator’ template has nodes on the *incorrect operation/comparison* fault (Fig. 13).

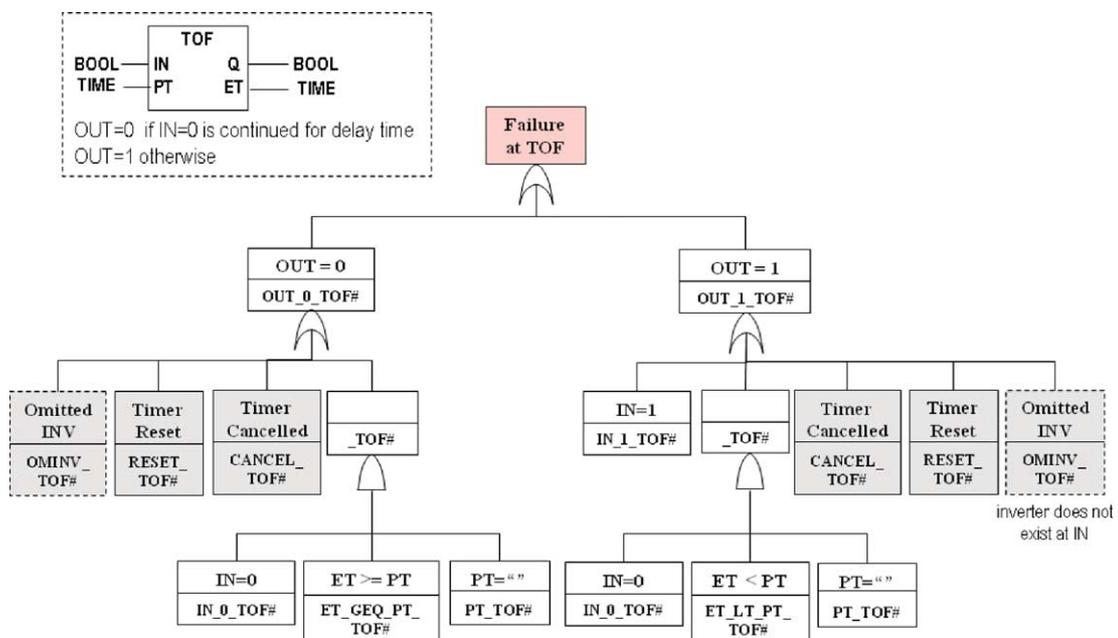


Fig. 12. TOF template.

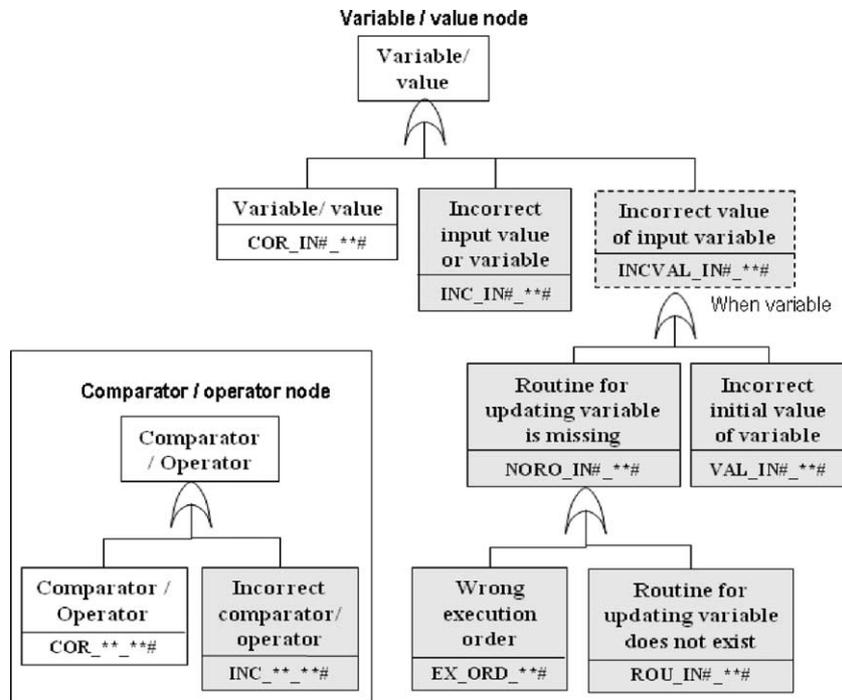


Fig. 13. Terminal node templates.

(8) *Others.* Converter blocks, inverters and the MOV block, which is a function that assigns an input value to an output variable, are added as a single node in the fault tree whenever it appears in the FBD.

5. Fault tree analysis procedure and case study

In this section, we give a procedure for the combined template-based fault tree analysis with a case study applied on a partial RPS design.

Step 1. Identification of hazard and related FBDs. First, the undesired event of the system is determined and the corresponding output in the FBD is identified. Next, all networks that can contribute to this output are identified.

Step 2. Fault tree generation. The top event template is put at the top of the fault tree with the undesired event as the top event, and the templates of the blocks directly connected to the output block are attached. Each branch is expanded until there are no dependent FBD routines or function blocks left. When all templates are attached, terminal node templates are added to the remaining cause nodes that are leaf nodes of the generated tree. ‘variable/value’ templates are attached to the value or variable cause node whereas the ‘comparator/operator’ templates are attached to the operator or comparator nodes.

When expanding fault trees, analyst may choose to simplify the fault tree by eliminating irrelevant branches. For example, if an unwanted event occurred at the output of an AND block by outputting an incorrect value 1, only

the rightmost subtree, the subtree with top event ‘Output = 1’ may be attached as shown in Fig. 14.

Step 3. Cut-set analysis. The last step is to generate the minimal cut sets, as typically performed on safety analysis, so that analysts may obtain additional insights as to how logical design errors found through fault tree analysis can be best corrected.

5.1. Case study

This section shows how the proposed approach was applied to the safety analysis of RPS of the nuclear power plant system, which is currently being developed at KNICS [4] in South Korea. It is also compared to the fault tree manually developed by a domain expert.

The *manual reset variable set-point trip with operating bypass*, whose partial logic is shown in Fig. 15, is a trip logic in nuclear power plant’s Digital Plant Protection System (DPPS) Reactor Protection System (RPS) Bistable Processor (BP). While the input value (log power: f_X) is valid ($f_{X_Valid}=1$), if the input value falls below the predefined fixed trip set-point then the trip signal (th_{X_Trip}) occurs. A preliminary trip logic ($th_{X_Pre-trip}$) exists that has a higher trip set-point than the trip, which plays a role in warning the operator beforehand. If a pretrip signal occurs, the operator can manually reset the set-point before a trip signal occurs. This way, the set-point keeps falling, allowing the system to shutdown safely. A trip bypass logic ($h_{X_OB_STA}$) shows that a trip does not occur if the operator initiated a bypass signal.

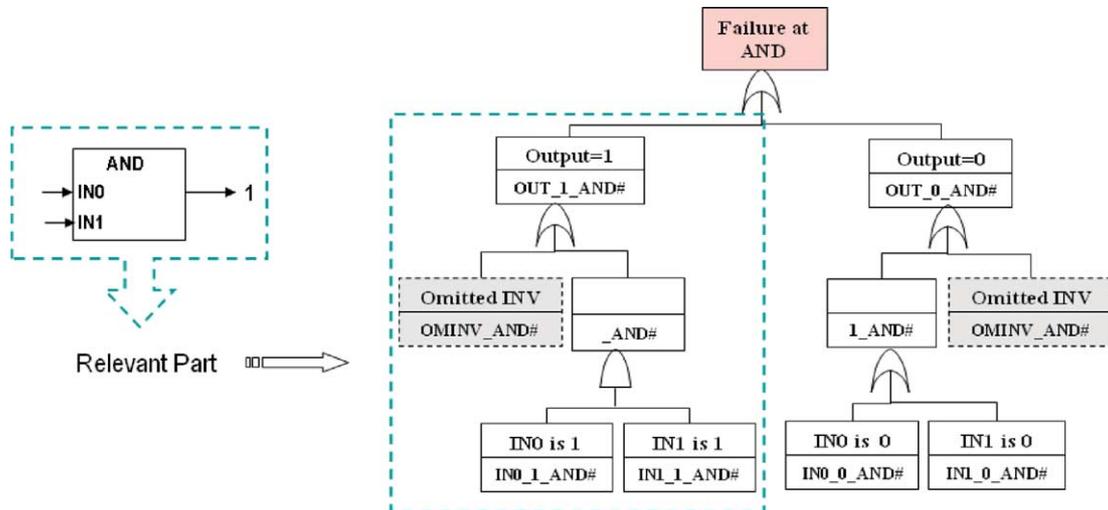


Fig. 14. Partial application of AND template.

Step 1. Identification of hazard and related FBDs. The FBD of the RPS is designed so that the trip signal corresponds to the variable th_X_Trip which is the output of the FBD shown in Fig. 15, and the occurrence of the trip signal is depicted by ' $th_X_Trip=0$ ' and ' $th_X_Trip=1$ ' otherwise. We identify the unwanted event as 'normal signal (output 0 or 1) occurs but outputs the wrong signal.' Since the FBD for RPS is based on a 0/1 logic, the top event of the fault tree is set to 'Incorrect output of 0' or 'Incorrect output of 1'. The top event for the fault tree of the trip logic in Fig. 15 is 'incorrect output of $th_X_Trip=1$ ', which means that a signal of ' $th_X_Trip=1$ ' occurs although the system behavior was intended for a trip signal to occur. The dependent networks in Fig. 16 are $h_X_OB_STA$ and f_X_Valid routines.

Step 2. Fault tree generation. This step is shown in Figs. 16 and 17. The incorrect output ' $th_X_Trip=1$ ' (failed to output trip signal) becomes the top event. The fault tree generation begins by applying the top event template as in Fig. 16. After the top event template is attached, the AND template of the output block, AND1, is attached. The AND templates of the following AND2 and AND3 blocks are also attached. Although all are AND functions, the attached templates look somewhat different in Fig. 16. This is because in AND2 and AND3, inverters are added as single nodes, and in AND3, the *Omitted INV* node is not attached since inverters exist at both of the inputs of the AND3 block.

The templates for all following blocks are attached in the same manner. Once all the templates of the blocks in all dependent routines have been applied, the terminal node templates are attached to the leaf nodes to show the possibilities of incorrect inputs, routine faults or wrong operations and comparisons. In Fig. 17, the terminal node templates are attached to the cause leaf nodes of the LE block because there are no following connections to it.

Step 3. Cut-set analysis. Once the fault tree is generated, it will contain all possible sets of events that can contribute

to the hazardous result. The analyst can follow the paths where the nodes are 'true' and will eventually come upon a set of events that correspond to the current FBD design. The fault nodes in this set are the faults that caused the top event to occur. In this case, we found a cut set starting from the top node FAIL_TRIP ($th_X_Trip=1$) continuing down to leaf nodes. The leaf nodes contained one fault node, which was an 'incorrect input value' node. This node describes that the input of IN1, which is being compared to IN0 by LE1 (less than or equal), was incorrect. Therefore we can conclude here that the input of IN1, $k_X_Trip_setpoint$ is incorrect. Input IN1 should be the variable $h_X_Trip_Setpoint$, and not the fixed value $k_X_Trip_Setpoint$.

5.2. Comparative study

A fault tree was manually prepared by domain experts who have extensive experience on applying fault tree analysis and familiar with features of the RPS system used in the case study. Since it captures only the likely causes to the same hazard in the experts' personal opinion, it would vary from one expert to another. Fig. 18 shows a part of the fault tree for Fig. 15. The left tree is the manual fault tree by safety engineers and the right tree is the fault tree generated by our approach.

The dotted boxes show some of the differences between the two fault trees. Since the undesired output is ' $th_X_Trip=1$ ', all inputs to AND1 are 1 making inputs to AND2 and AND3 1 also (0 when there is an inverter on the input line). The bottom nodes on both trees contain this information.

In the manually generated tree, errors such as misplaced inverters or omitted inverters are not considered. Although these are some of the most frequently made errors in the FBD design, the safety analyst assumed that the inverters are correctly placed. *Missing input* is

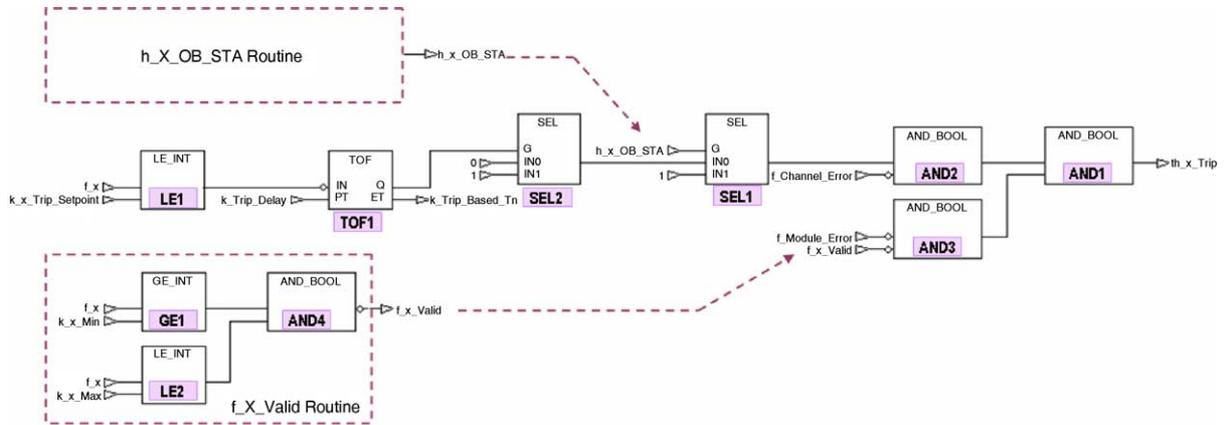


Fig. 15. FBD of manual reset variable set-point trip with operating bypass logic.

also missing in the manually developed fault tree and not considered as a fault.

Faults such as *wrong execution order* or *missing routine* are not consistently considered for all variables either. For example, *wrong execution order* and *missing routine* are

considered for the variable $h_X_{OB_STA}$ (not shown in Fig. 18). However, they are not considered for the variable f_X_{Valid} , as can be seen in Fig. 18 that the node ' $f_X_{Valid}=0$ ' is not developed further, although a routine for it does exist in the design. In the manual fault tree,

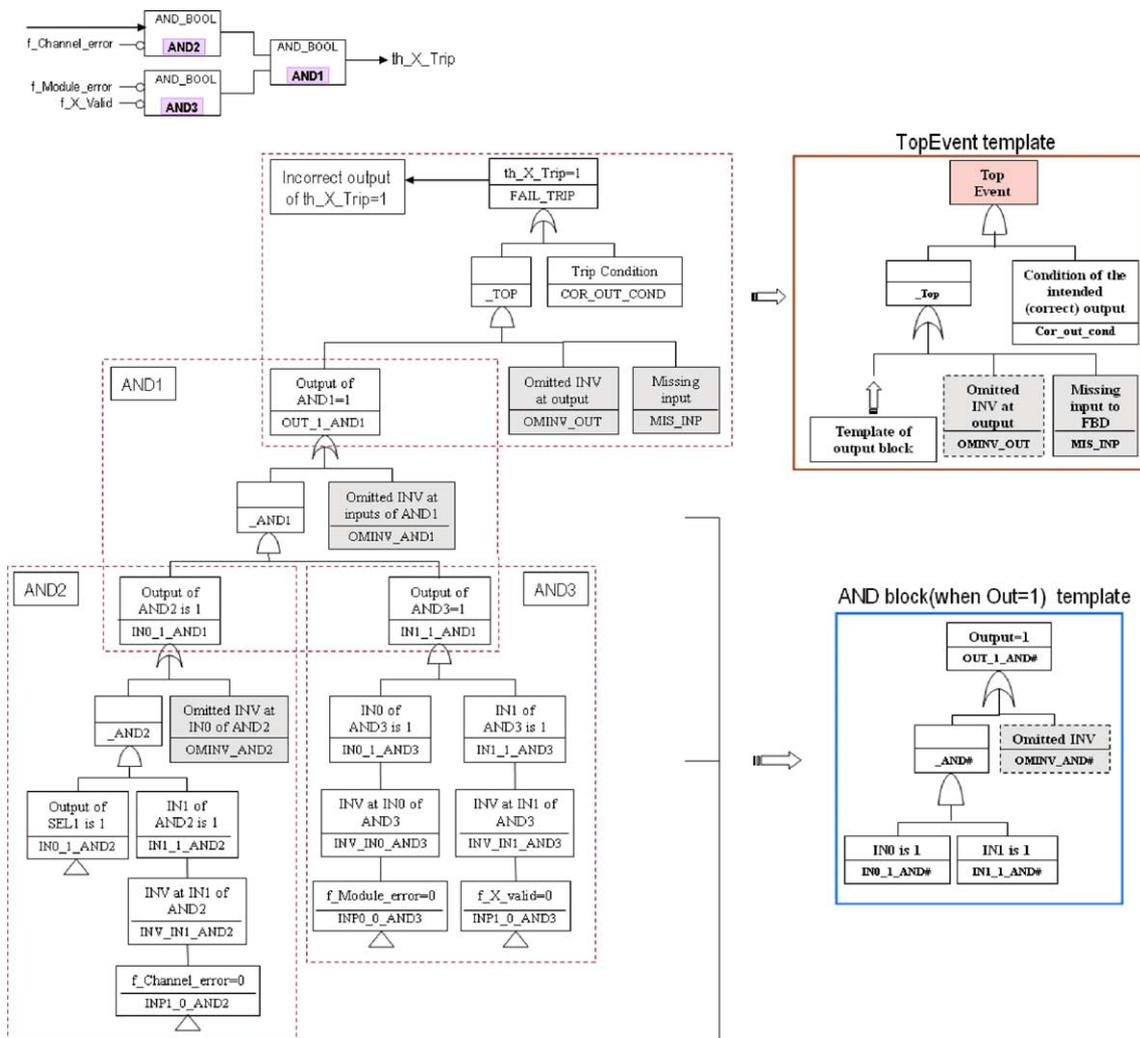


Fig. 16. Fault tree generation.

internal block errors are considered to contribute to the undesired output but faults such as *incorrect input variable or value* are not explicitly shown. Since the fault tree analysis depends on the analysts, the analysts' knowledge or assumptions will affect how complete the analysis becomes. When two different fault trees are compared for completeness and comprehensiveness, the following observations can be made:

- (1) Proposed technique is clearer and easier to understand because the propagation path events (causes) and fault events are separately depicted.
- (2) Faults that are frequently missed in manual analysis (e.g. omission faults) are explicitly identified.

- (3) As the proposed approach provides templates to be used in the fault tree generation, safety analysis is likely to be more productive than the manual approach.
- (4) Templates contain complementary viewpoints and conditional fault nodes. Therefore, it is highly likely to be more complex (e.g. more nodes and branches) than the manually developed and 'focused' fault tree.

The fault tree was proved to be useful by nuclear domain experts in the safety analysis of FBDs used in the nuclear domain. Safety expert's opinions on the combined template-based approach were that it provided a more complete analysis in identifying faults and that the templates provide a semi-automatic fault tree analysis that is easy to apply.

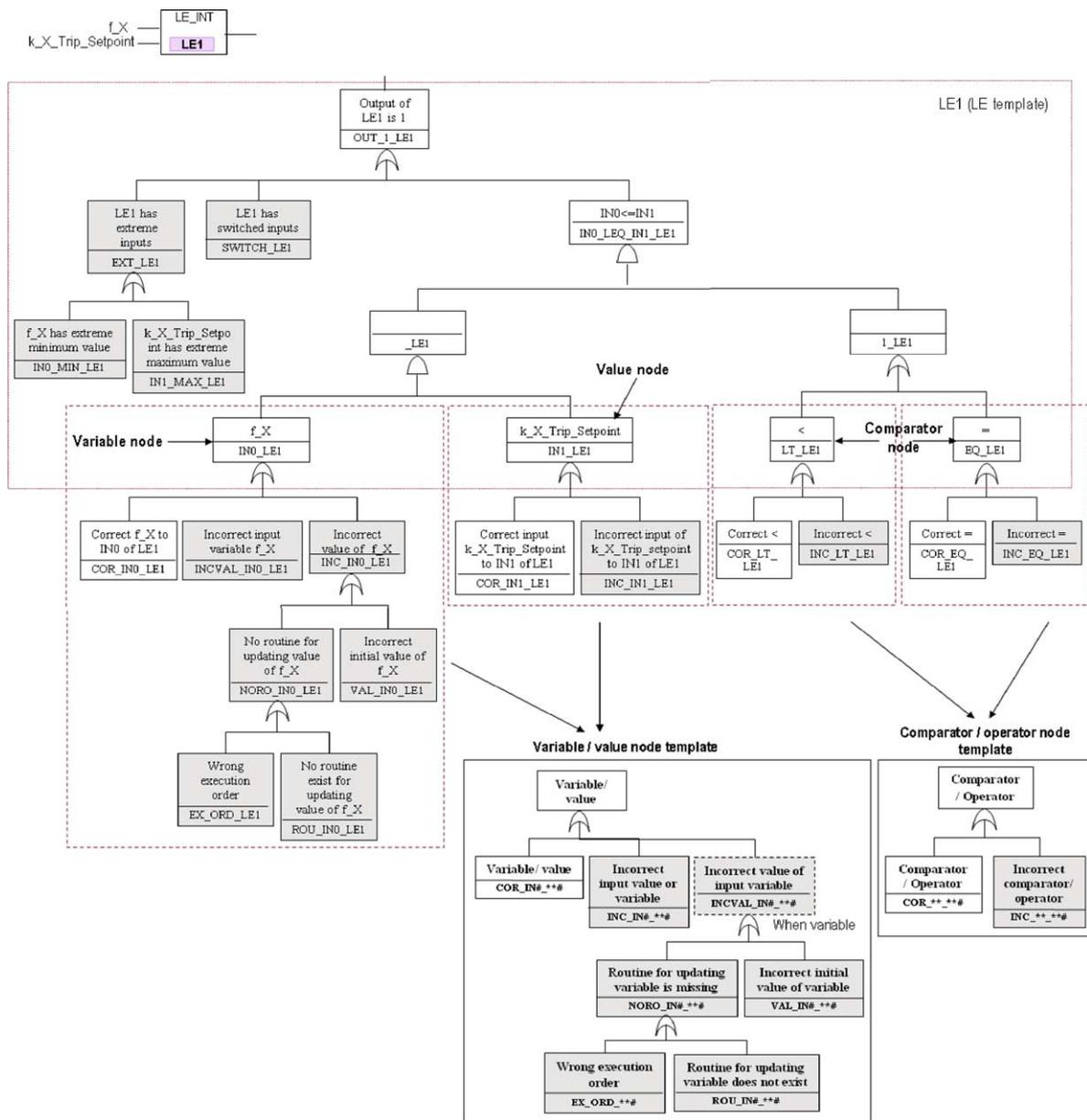


Fig. 17. Application of terminal node templates.

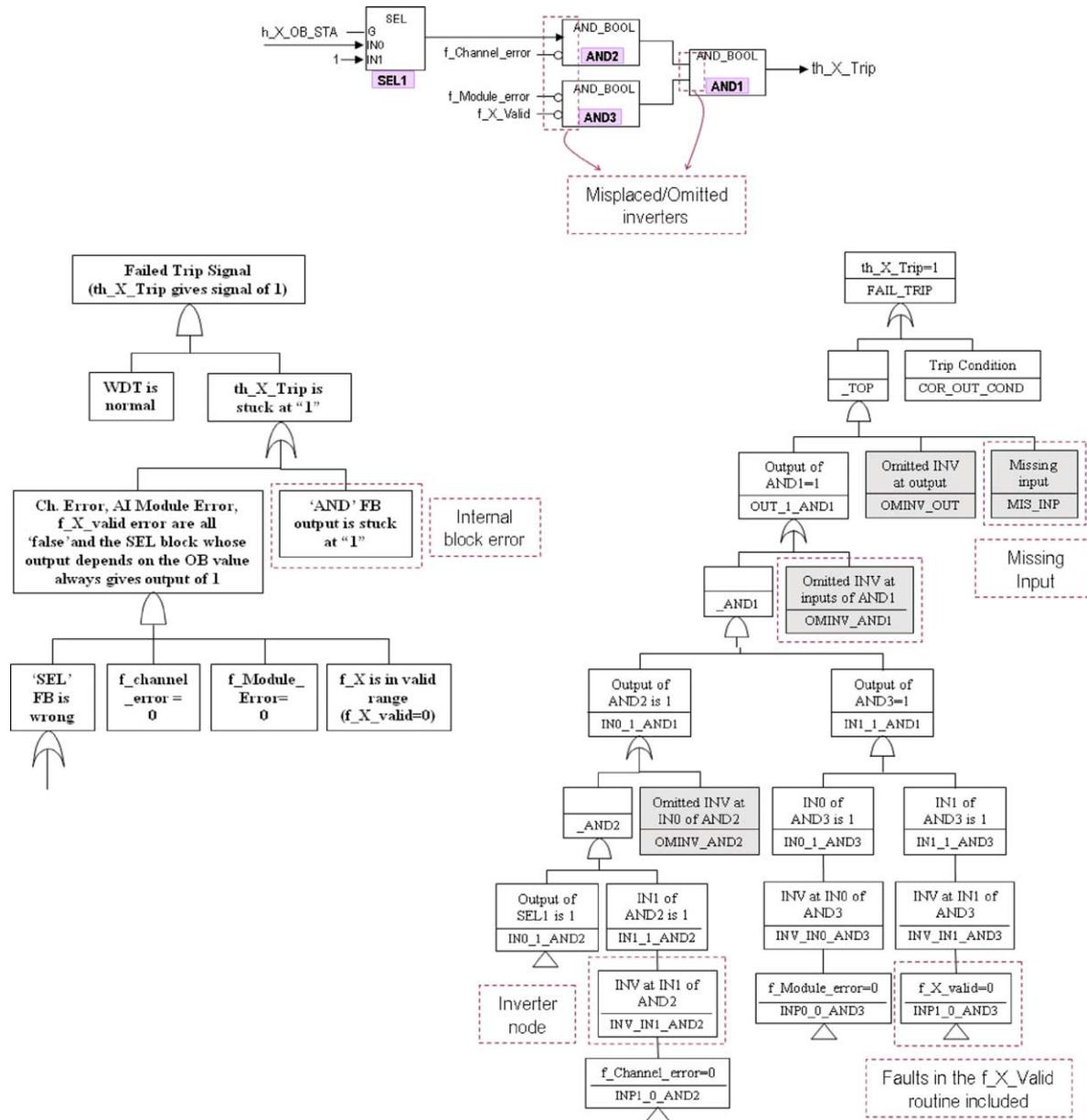


Fig. 18. Comparison of two fault trees generated by different approaches.

6. Conclusion and future work

In this paper, we propose a fault tree analysis technique on function block diagrams. As FBDs are often used to implement safety-critical software, such techniques are needed to achieve desired quality assurance and to satisfy regulatory requirements. Our approach uses templates combining *fault-oriented* and *CE-oriented* views so that safety analysis may be more thorough and easier to review for completeness. The *fault-oriented* view contains information on the possible faults that can occur in FBDs. The faults were categorized and included in the templates. The *CE-oriented* view contains the functions of each function

block. Since FBD is a connection of blocks, the function blocks that produce an output become the propagation paths of the faults. Fault tree templates combined from the two viewpoints help analysts perform more comprehensive safety analysis than a manual approach.

Our technique was applied to the representative trip logic of KNICS RPS, which is currently being developed in South Korea, and it shows that it is applicable to real-world systems. Nuclear engineers found the proposed template-based fault tree analysis approach was proved to be useful in identifying faults leading to the undesired trip result in RPS. A future research plan is to make the templates more context-sensitive to the FBD specification in order for the generated fault tree to be more compact.

References

- [1] Leveson N. *SAFWARE: system safety and computers*. Reading, MA: Addison-Wesley; 1995.
- [2] International Electrotechnical Commission. *IE International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages*; 1993.
- [3] Mader A. A classification of PLC models and applications. In: *Proceedings of WODES 2000: fifth workshop on discrete event systems*, Gent, Belgium; August 21–23 2000.
- [4] KNICS. Korea Nuclear Instrumentation and Control System Research and Development Center. <http://www.knics.re.kr/english/eindex.html>
- [5] Maier T. FMEA and FTA to support safe design of embedded software in safety critical systems. In: *Proceedings of CRS 12th annual workshop on safety and reliability of software based systems*; 1997, p. 351–67.
- [6] Mauri G. *Integrating safety analysis techniques, supporting identification of common cause failures*. PhD Theses; 2000.
- [7] Vesely W, Goldberg F, Roberts N, Hasel D. *Fault tree handbook*. System and Reliability Research Office of Nuclear Regulatory Research, US Nuclear Regulatory Commission; January 1981.
- [8] Andrews JD, Brennan G. Application of the Digraph method of fault-tree construction to a complex control configuration. *Reliab Eng Syst Safety* 1990;28(3):357–84.
- [9] Kocza G, Bossche A. Automatic fault-tree synthesis and real-time trimming, based on computer models. In: *Proceedings of reliability and maintainability symposium*; January 1997, p. 13–6.
- [10] Garret C, Guarro S, Apostolakis G. The dynamic flowgraph methodology for assessing the dependability of embedded software systems. *IEEE Trans Syst, Man Cybernetics* 1995; 25(5):824–40.
- [11] Yau M, Guarro S, Apostolakis G. Demonstration of the dynamic flowgraph methodology using the Titan II space launch vehicle digital flight control system. *Reliab Eng Syst Safety* 1995;49(3):335–53.
- [12] Yau M, Apostolakis G, Guarro S. The use of prime implicants in dependability analysis of software controlled systems. *Reliab Eng Syst Safety* 1998;62(1):23–32.
- [13] Garrett C, Apostolakis G. Automated hazard analysis of digital control systems. *Reliab Eng Syst Safety* 2002;77(1):1–17.
- [14] Cha S, Leveson N, Shimeall T. Safety verification in murphy using fault tree analysis. In: *Proceedings 10th international conference on software engineering*, Singapore; April 1988, p. 377–86.
- [15] Leveson N, Cha S, Shimeall T. Safety verification of Ada programs using software fault trees. *IEEE Software* 1991;8(4):48–59.
- [16] Min SY, Jang YK, Cha S, Kwon YR, Bae DH. Safety verification of Ada95 programs using software fault trees. In: *Proceedings of computer safety, reliability and security: 18th international conference, SAFECOMP'99*, Toulouse, France; September 1999, p. 226–38.
- [17] Papadopoulos Y, McDermid J, Sasse R, Heiner G. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliab Eng Syst Safety* 2001;71(3): 229–47.
- [18] Papadopoulos Y, Maruhn M. Model-based synthesis of fault trees from Matlab–Simulink models. In: *Proceedings of DSN 2001*; 2001, p. 77–82.
- [19] Vemuri K, Dugan J, Sullivan K. Automatic synthesis of fault trees for computer-based systems. *IEEE Trans Reliab* 1999;48(4):394–402.
- [20] Sullivan K, Dugan J, Coppit D. The Galileo fault tree analysis tool. In: *Proceedings of the 29th annual international symposium on fault-tolerant computing*; June 1999, p. 232–35.
- [21] Li B, Li M, Ghose S, Smidts C. Integrating software into PRA. In: *Proceedings of 14th international symposium on software reliability engineering, ISSRE 2003*; November 2003, p. 17–20.