Reliability Engineering and System Safety 203 (2020) 107064

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

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

# A Software Fault Tree Analysis Technique for Formal Requirement Specifications of Nuclear 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

ARTICLE INFO

*Keywords:*
Software Safety Analysis
Software Fault Tree Analysis
Requirements Specification
Formal Specification

ABSTRACT

Rigorous safety demonstration of software, as well as systems, is required when developing digital reactor protection systems in nuclear power plants. Various safety analysis techniques try to identify, analyze, and find remedies for potential hazards at each stage of software development life-cycle. This paper proposes a software fault tree analysis technique for software requirements written in the NuSCR formal specification language. The proposed method can mechanically construct software fault trees and calculate minimal cut-sets, encompassing timing constraints of multi-cycles, from NuSCR formal specifications. We also improved the fault tree construction and analysis tool into "*NuFTA 2.0*" to cope with multi-cycled executions. The case study with a preliminary version of requirements specification for a Korean nuclear power plant in operation shows the proposed technique's effectiveness and applicability to other V&Vs such as simulation.

## 1. Introduction

Rigorous safety demonstration of software through safety analysis is highly recommended by government authorities [2,3] and international standards [4–6], when developing nuclear reactor protection system (RPS). Various safety analysis techniques such as fault tree analysis (FTA), failure mode and effect analysis (FMEA) and Hazard and operability (HAZOP) try to identify potential hazards in software and systems, analyze their cause and effect, and then propose remedies/solutions to avoid/overcome identified potential hazards. Software safety analysis techniques [7,8] are also recommended to apply at each stage of software development life-cycle [9,10] as shown in [11,12].

FTA is one of the most widely used safety/hazard analysis techniques [13,14], and its application to software is often called software fault tree analysis (SFTA). SFTA is used to detect software logic errors, identify conditions that need to initiate fail-safe and fault tolerance mechanism, and generate effective test cases. SFTA techniques have been used in various software development phases and several domains by manually or automatically. For example, [15–17] proposed SFTA techniques for safety-critical software and systems in nuclear power plants (NPP). They constructed software fault trees mechanically or used fault tree templates to construct them from software requirements/design specifications or codes. Fault tree analysis with unified modelling language (UML) such as generating fault trees from UML use case [18] or activity diagram [19] is also an example of SFTA. We have also proposed a SFTA technique - '*NuFTA*' [20], which can mechanically generate fault trees and refined logic formula from a software requirement specifications written in *NuSCR (Nuclear Software Cost Reduction)* [21] formal specification language.

Software in nuclear RPSs encompasses the timing-constrained requirements of multi-cycles, which should be well considered to define important system operations such as the shutdown of nuclear reactors [22]. While such systems like NPPs operate under continuous timing constraints, previous researches, including NuFTA, does not take into account the timing constraints of multi cycles precisely. This paper proposes a refined SFTA technique that resolves two technical challenges left by our prevous work. We also implemented it into a supporting tool "*NuFTA 2.0*". (1) Our former work could deal with only one execution cycle of a whole software system, due to its theoretical basis [16]. We defined new fault tree semantics with respect to the timing constrains of multi-cycles, and proposed a new set of fault-tree templates. *NuFTA 2.0* now constructs a fault tree of 100 ~ 200 execution cycles without being overwhelmed by unnecessary information. (2) The previous version of *NuFTA* produced too large unrefined logic formula (similar to *cut-sets*) to analyze. We proposed a new fault tree construction algorithm based on set theory [23]. *NuFTA 2.0* now constructs a fault tree of 2,389 nodes in 0.228 seconds, and also produces well-refined minimal cut-sets (MCSs) in 1.037 seconds. The MCSs, which

☆ A preliminary version of this paper was published in *KNS Autumn Meeting 2016*[1]

* Corresponding author.
*E-mail address:* jbyoo@konkuk.ac.kr (J. Yoo).

https://doi.org/10.1016/j.ress.2020.107064
Received 3 September 2019; Received in revised form 27 May 2020; Accepted 3 June 2020
Available online 10 June 2020
0951-8320/ 

take into account multi-cycled timing constraints of the software requirements specifications, can also be used for generating simulation scenarios in design or implementation phases.

We performed a case study on a preliminary version [24] of an RPS in a Korean nuclear power plant, it showed that the proposed SFTA technique and the supporting tool, *NuFTA 2.0*, can help safety analysts to analyze the RPS software more efficiently and precisely, it also helps developers prepare simulation scenarios mechanically. The remainder of the paper is organized as follows: Section 2 briefly overviews the SFTA techniques, and introduces the *NuSCR* formal requirements specification language to aid understanding of the software fault tree (SFT) templates proposed in the paper. Section 6 explains the new SFT templates, the new SFT construction algorithm, and the multi-cycled MCSs calcuation algorithm. Section 4 describes the new supporting tool *NuFTA 2.0*, and we explain the case study in Section 5. Section 6 concludes the paper and gives remarks on our future research direction.

## 2. Background

### 2.1. Software Fault Tree Analysis

FTA [25] is a top down, deductive failure analysis in which an undesired state of a system is analyzed using Boolean logic to combine a series of lower-level events. SFTA, whose target is software, is one of the softwrae safety analysis techniques trying to detect software logic errors, determine the conditions under which fault tolerance and fail-safe procedures should be initiated, and facilitates effective safety testing by pinpointing critical test cases [26]. The SFTA technique can be applied at any stage of the software development lifecycle, such as requirements analysis, design or coding. SFTA is often done qualitatively, while traditional FTA can be done either qualitatively or quantitatively, due to the existence of software-related failure data [27,28]. SFTA is typically hard and unavailalbe to estimate probabilities of failures without probability information of the software elements. Functional safety standards such as IEC-61508[6] and ISO-26262[29] use a predefined failure probability just to define the safety integrity level of software elements not to do analyze with quantitatevely.

### 2.2. NuSCR

NuSCR [21] is a formal software requirements specification language, targeting for safety-critical control software in nuclear power plants. It extends software cost reduction (SCR) [30] with `finite state machine (FSM)` and time-based notation in addition to decision tables in order to reduce the complexity of formal specifications. NuSCR is also a corner-stone of the "*NuDE 2.0*," a formal method-based software development, verification, and safety analysis environment for safety-critical digital instrumentation and control system (I&Cs) that is implemented with a programmable logic controller (PLC) and field-programmable gate array. It is a model-based development framework for nuclear domain. More than 20 case studies were performed by *NuDE 2.0* as summarized in [31,32]. ⟨Fig. 1(a) ⟩ shows the NuSCR modeling tool - "*NuSRS 2.1*," which is one of 13 tools in *NuDE 2.0*. We write '*HI_LOG_POWER*' as 'X' in ⟨ Fig. 1 (D)⟩ for improving readability.

⟨ Fig. 1(b) ∼ (d))⟩ illustrate a part of the *NuSCR* specification for *g_HI_LOG_POWER*, which is *fixed set-point rising trip* logic, in KNICS RPS bistable process (BP) [24]. *NuSCR* is basically based on the *Parnas' Four-Variable Model* [33] and additionally uses three other basic constructs such as *function variable, history variable*, and *timed history variable* to specify functionalities easily and each variable is represented by a variable node called the `structured decision table (SDT)`, `FSM`, and `timed transition system (TTS)` [34], respectively. The relationship of all constructs is expressed by `FOD` (Function Overview Diagram) which is a kind of data-flow diagram. The `FOD` allows hierarchical modeling of arbitrary depth. ⟨Fig. 1(b)⟩ shows the `FOD` for the *g_HI_LOG_POWER* module and ⟨Fig. 1(c)⟩ and ⟨Fig. 1(d)⟩ are the `SDT` and `TTS` nodes in from in the ⟨Fig. 1(b)⟩.

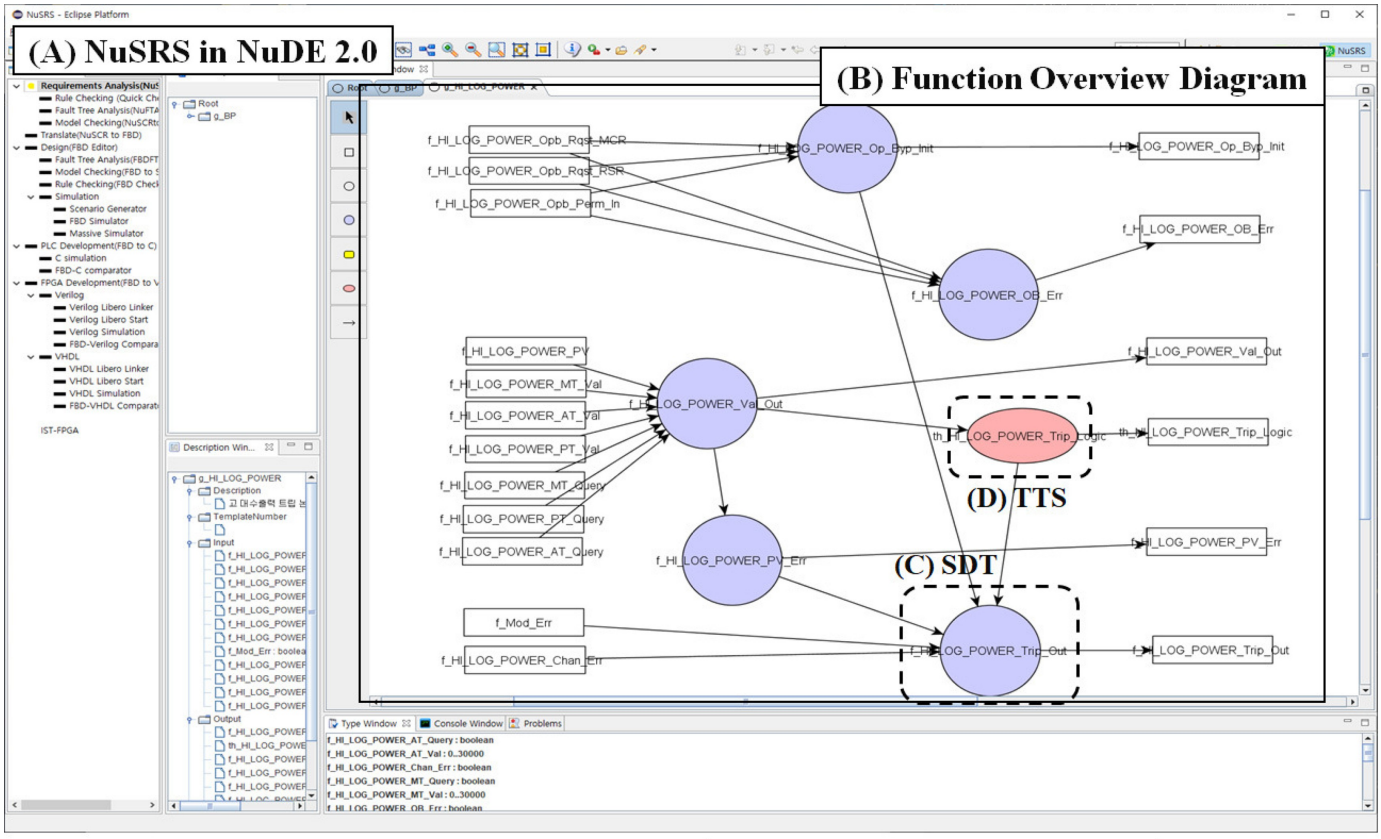
⟨ Fig. 1(c)⟩ is a `SDT` for the function variable node *f_HI_LOG_POWER_Trip_Out* in the FOD. It decides a final output of the logic according to the value of *th_HI_LOG_POWER_ Trip_logic* and 3 incoming errors. ⟨Fig. 1(d)⟩ is a `TTS` definition for the timed history variable node *th_HI_LOG_POWER_Trip*. `TTS` is an `FSM` extended with a timing constraint such as [a, b] onto transitions. It is interpreted as follows: "If the condition '*f_HI_LOG_POWER _Val_Out > k_HI_LOG_POWER_Trip_Set*' is satisfied in state '*Normal*', it transits to '*Waiting*' state. In this state, if the conditions '*f_HI _LOG_POWER_Val_Out > k_HI_LOG _POWER_Trip_Set*' lasts for *k_HI_LOG_POWER_ Trip_Delay* time, then it fires the trip signal (*th_HI_LOG_POWER_Trip: = true*) and transits to the '*Trip*' state." The `TTS` expressions [k_HI_LOG_POWER_Trip_Delay, k_HI_LOG_ POWER_-Trip_Delay] means that the condition has to remain *true* for k_HI_-LOG_POWER_ Trip_Delay time units. A history variable in NuSCR is represented by a history variable node in `FOD` and is defined by `FSM`. A detailed explanation of the `FSM` is skipped because of the `FSM` is defined same as `TTS` except for the timing constraints.

### 2.3. Related Work

Safety and reliability analysis of software has been studied several ways. Vyas et al. [28] presented a literature review of software FMEA and FTA along the software development life-cycles. Almost all SFTA of software requirement analysis phases, as mentioned in the [28], they have been performed manually on use cases, textual descriptions of use case, or natural language requirements. For example, eliciting safety requirement from use cases is proposed by [18], and it constructs a fault tree manually from UML use case. Tiwari et al. [35] proposed SFTA and SFMEA that use a formalized use case template, but the descriptions of its contents are written in natural language. There are also several software fault tree constructions and analysis techniques for requirements specification that are semi- or fully- automatically from formal specification such as using Statecharts [36], RSML [37], and NuSCR [16]. SFTA in the requirement analysis phase is used to identify the safety-related faults in specifications or missing/weakness requirements.

At software design phase, SFTA is applied to design specifications, such as the function bloack diagram (FBD) [38], UML, Etc. SFTA for the design phase is more automated than for the requirement analysis phases. Lauer et al. [39] introduced automatically synthesis algorithms for generating fault trees from UML model at a design level. Dickerson et al.[19] proposed a foraml transformation method for generating fault trees from an UML activity diagram with definitions of semantic mapping rules. Oveisi et al. [40] proposed an SFTA-based approach for analyzing operational use cases in cyber physical systems. It automatically constructs a fault tree from use case descriptions, however, it only considers events of software system defined in use cases. In the [15,17], the authors defined fault tree templates for FBDs, and applied them to the shutdown logic for an RPS in Korea to analyze logical correctness of the FBD designs. [41] proposed a tool and method to construct software fault tree efficiently by reusing SFTA information, however, fault tree construction is based on manual generation firstly.

Generating fault trees from models is also one of issues for safety analysis. Kabir [42] reviewed the standard fault tree analysis methods and some of its exetensions, such as dynamic, state/event, or component fault trees, the author also invenstigated a number of prominent model based dependability analysis techniques that use fault trees. Many researches introduced in [42], focus on system-level analysis for reliability and dependability analysis, and software requirement level




| Conditions | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| th_HI_LOG_POWER_Trip_Logic = true | T | T | T | F | F | F | T | F |
| f_HI_LOG_POWER_Op_Byp_Init = false | T | T | F | T | F | T | F | F |
| f_Mod_Err = true \| f_HI_LOG_POWER_Chan_Err = true\| f_HI_LOG_POWER_PV_Err = true | T | F | T | T | T | F | F | F |
| **Actions** | | | | | | | | |
| f_HI_LOG_POWER_Trip_Out := true | O | O | O | O | O | | | |
| f_HI_LOG_POWER_Trip_Out := false | | | | | | O | O | O |

**Fig. 1.** A part of the *NuSCR* specification for the *g_HI_LOG_POWER* module in *NuSRS* (ver. 2.1)

analysis is less considerations. Generating fault trees from system modelling language (SysML) diagram [43,44] and generating component fault tree by using the architetural layers [45] are several studies for fault tree analysis by automatic generation. [46] proposed a method to generate dynamic fault trees automatically from system models that are internal block diagram and block definition diagram, with considering redundancy profile for systems. [46] also introduces several papers about generating fault trees from system models. Alshboul et al. [47] proposed an automatic derivation method of fault trees from SysML, it uses behavior and state diagram of the SysML with failure informatio to generate fault trees. These automatic generation method of fault trees are concerned to behavior model of system-level.

Tiwari and Gupta [48] proposed an approach for generating test cases from fault trees and software success tree. They constructed software success tree and SFT from UML activity diagram with mapping rules, and MCSs, which are created from the trees, are used to generate test cases for testing normal/exceptional behavior. Li et al. [49] introduced a use of fault tree analysis to software testing for improving quality and efficiency of the testing. The authors distincted importance elementary events by quantitative analysis using probability of each nodes, which represents a basic event of software. Philip et al. [50] proposed a framework that is automatic generation of safety validation test cases from architecture analysis and design language (AADL) extended model for software system. The framework of [50] constructs fault trees from AADL model and fault model, and generate test cases at system level, using minimal cut-sets and safety properties. Yadav et al. [51] proposed a model to predict the software defect density indicator of early phases of software development life-cycles. Several software

metrics of requirement analysis and design phases of software development is used and selected to predict reliability with fuzzy inferences as a quantitative analysis. Sinha et al. [52] proposed a model to predict reliability/availability of hardware-software combined system at early design phases. The authors identified functional requirements and constructed component/software behavior model. A simulation of the system behavior is used to predict reliability and availability of the system [52]. [53] introduced basic information of software reliability engineering such as reliability prediction model, reliability measurement, management. Software reliability analysis need to probability and failure information for predicting or measurement.

There are many approaches to fault trees for software and systems, in addition there are several extensions such as dynamic fault trees, state event fault trees. A method of generating fault trees from models that are system layer, SysML, or UML have been also studied with much attention. A lot of studies on fault tree analysis, including SFTA, focus on failure or reliability analysis of behavior of the system/software. A fault tree analysis of software that contains timing-constrained requirements of multi-cycles in software requirements level is less studied before. The proposed method and template in this paper can automatically generate fault trees with considering defined logics of NuSCR software requirements specification and containing multi-cycled conditions as a qualitative analysis. The MCSs, generated by proposed method in this paper, contains information and behavior of state transitions (*i.e.* sequence of behavior) according to the time steps as defined in software requirement specification and it can be used for simulation-based testing by generating scenarios. A detailed explanation and application of the proposed method and template is explained in next section.

## 3. The Software Fault Tree Analysis for *NuSCR* specifications

This section introduces a SFTA technique for software requirements specification written in NuSCR. It mechanically generates a set of software fault trees from formal specification, it also generates well-refined minimal cut-sets that encompass the timing constraints of multi-cycled executions. Section 3.1 proposes the fault tree templates used to construct SFTs mechanically from NuSCR formal requirements specifications. Section 3.2 explains the SFT construction algorithm which uses the SFT templates. Section 3.3 shows the minimal cut-sets calculation algorithms in the SFTs.

### 3.1. The Software Fault Tree Templates

A NuSCR formal requirements specification consists of 3 constructs defined as `SDT`, `FSM` and `TTS`, respectively. Our previous work [16,20] proposed fault tree templates for these constructs, but we could apply them to software behavior of one execution only, which is too strict to perform SFTA exhaustively. In order to do SFTA for the behavior of multi-cycled executions, this paper re-defined the fault tree template for the timed-history variables defined as `TTS`. The new SFT template for `TTS` reflects software behavior of multi-cycles as well as in a single cycle. We also refined the SFT templates for `SDT` and `FSM` in order to incorporate additional information for multi-cycled executions.

#### 3.1.1. The SFT Templates for *SDT*

〈Fig. 2〉 is the software fault tree template for `SDT`. We summarized the explanation of each element of the template in 〈Table 1〉. The root node may have a specific value or a range of values, since a function

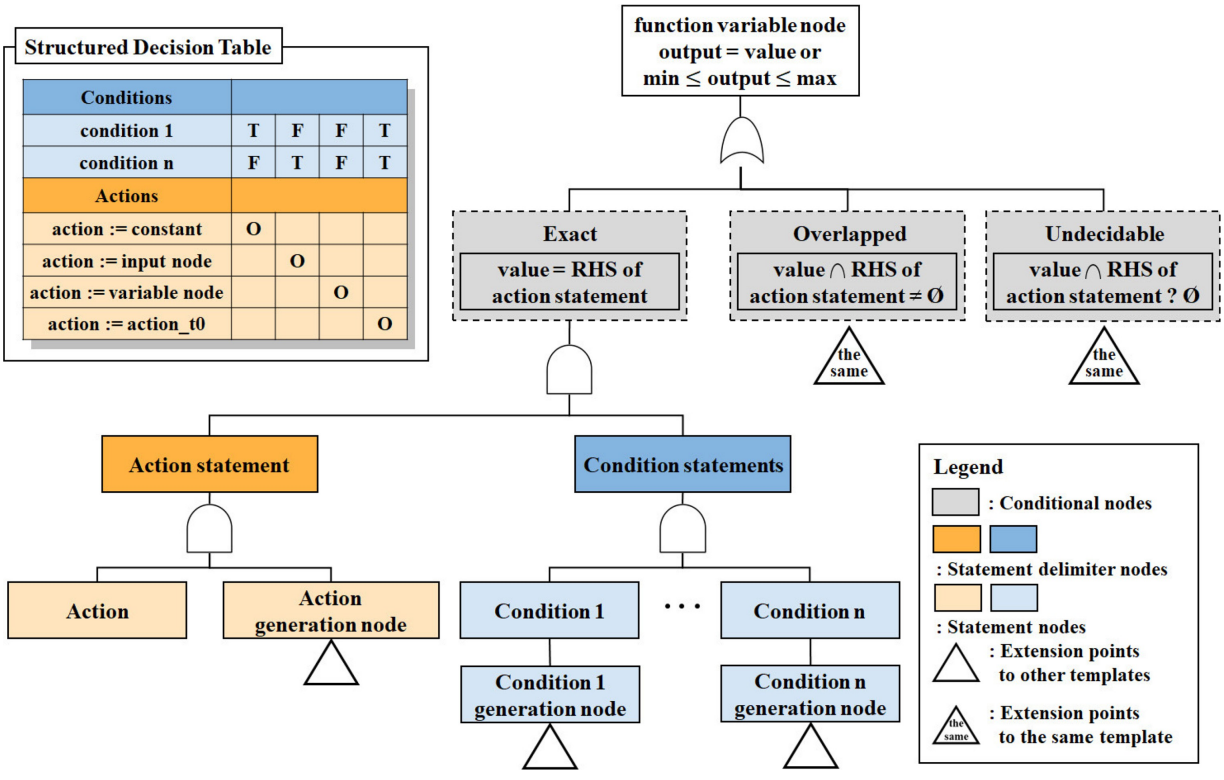



**Fig. 2.** An SFT template for `SDT`

**Table 1**
A description of the SFT template for SDT

| | | |
|---|---|---|
| Conditional Nodes | Exact | The output value of root node is the same value of the right hand side (RHS) of an action statement. |
| | overlapped | The output value of root node is overlapped with the RHS of an action statement. |
| | Undecidable | The exact relation between the output value and the action statement is undecidable. |
| Statement Nodes | Action Statement | An action corresponding to one row of SDT Actions, *i.e.,* possible output values of the node |
| | Condition Statements | A condition part corresponding to the rows of SDT Conditions, *i.e.,* action statements |

variable node is a kind of mathematical function (*i.e.,* a decision table). The nodes of dotted box are conditional nodes, indicating the value type of root node. All three types of sub-trees can have the similar templates. The action generation node and condition generation node means a decided output condition of action and condition nodes.

### 3.1.2. The SFT Templates for FSM

The history variable node is defined as an FSM consisting of the finite number of states, transitions and transition labels (*i.e.,* "*condition / action*"). The *NuSCR* FSM is a 'mealy-typed machine' [54], and the output is formed by both the state and the transition. It makes us difficult to derive an output value directly from an FSM, and we need to transform the FSM into the moore-type machine[55], whose output is decided by the state information only. We unfold the *NuSCR* FSMs into '*annotated FSMs*' where the configurations are clearly marked to be analyzed mechanically, by pairing a state and corresponding incoming transitions.

⟨ Algorithm 1 ⟩ shows the process of unfolding FSMs by traversing all transitions of all states in a depth-first strategy. If a transition has no action statement, it assigns the output value of the transitions's source state to the action statement of the transition, as line 3-5. Lines 6-12 also checks whether the annotated state that we want to create has already been created or not. If not, it creates a new annotated state and set the necessary information, as line 13-16.

⟨ Fig. 3 ⟩ is the software fault tree template for FSM node. We summarized its detailed explanation in ⟨Table 2⟩. The root node of the template may have a specific value, state, or range of values, since a history variable node is a kind of state-based mathematical model. The nodes of dotted boxes are conditional nodes, indicating the value type of the root node, and all types of sub-trees can have similar sub-templates. The nodes of pink box are case nodes, showing the case where the FSM is in the *state* at time *t*.

**Require:** $fsm$ : FSM, $annotatedState$ : an initial annotated state of FSM
1: **function** UNFOLDFSM($fsm$, $annotatedState$)
2: **for all** $Transition\ t \in annotatedState$.getOutgoingTransitions() **do**
3: **if** $t$.hasActionStatement() is false **then**
4: $t$.setActionStatement($annotatedState$.getActionStatement())
5: **end if**
6: **for all** $AnnotatedState\ tmpAS \in fsm$.getAnnoatedStates() **do**
7: **if** $checkExistingAnnotatedState(tmpAS, t)$ is true **then**
8: $tmpAS.setPreviousAnnotatedState(annotatedState)$
9: $exist \leftarrow true$
10: $break$
11: **end if**
12: **end for**
13: **if** $exist$ is false **then**
14: **Create a annotated state** $nextAS$
15: $settingAnnotatedState(nextAS, fsm, t, annotatedState)$
16: **unfoldFSM(**$fsm$**,** $nextAS$**)**
17: **end if**
18: **end for**
19: **end function**

**Algorithm 1.** Unfolding FSM

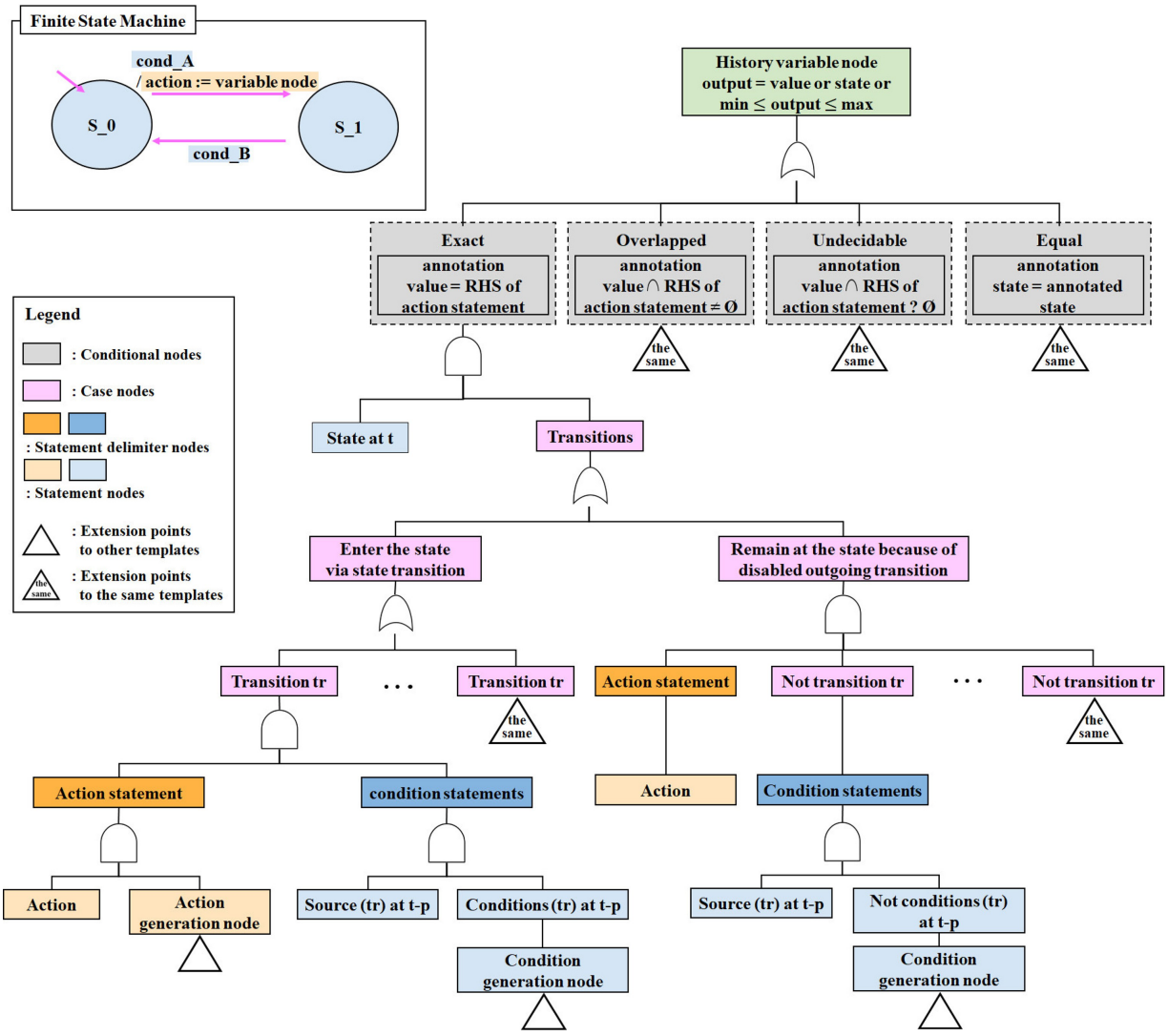



**Fig. 3.** An SFT template for FSM

**Table 2**
A description of the SFT template for FSM

| | | |
|---|---|---|
| Conditional Nodes | Exact | The output value of root node is the same value of the RHS of an action statement. |
| | Overlapped | The output value of root node is overlapped with the RHS of an action statement. |
| | Undecidable | The exact relation between the output value and the action statement is undecidable. |
| | Equal | The output state of root node is the same state of the history variable node. |
| Case Nodes | Entering the state | Entering the state through satisfying the condition of the ingoing transition |
| | Remaining | Remaining at the state because the conditions of all outgoing transition aren't satisfied |
| Statement Nodes | Action Statement | An action parts corresponding to the output of FSM's annotated state, *i.e.,* possible output values of the node |
| | Condition Statements | A condition parts corresponding to the transition, *i.e.,* conditions for satisfying the above case node |

### 3.1.3. *The SFT Templates for TTS*

The timed history variable node is defined as an TTS, which is an extension of FSMs with timing constraints. The template for TTS is very similar to the one for FSM, except for the subtree relating to timing constraints. The TTS is also unfolded to an '*annotated TTSs*' by the guide in ⟨Algorithm 1⟩. ⟨Fig. 4⟩ is the software fault tree template for the TTS node. The template for the TTS is fundamentally the same as for FSM, except that the time nodes are different. The time nodes are indicated with the yellow and show transitions containing a timing constraint. We summarized the template elements in ⟨Table 3⟩.

Time nodes can be seen as a sequence of behaviors at each discrete time unit. When the satisfaction time is the specific time that the condition has to remain *true*, the behavior is as follows:

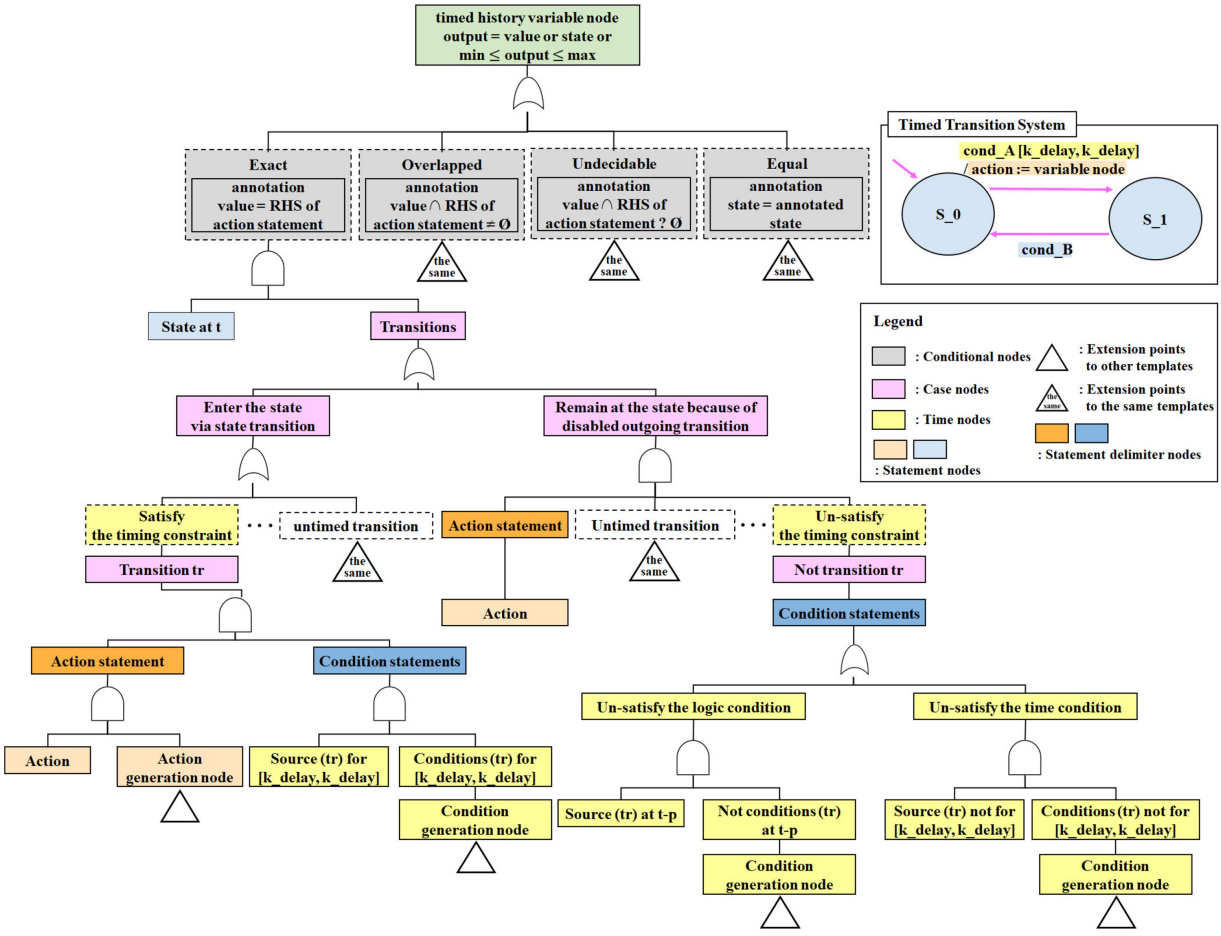



**Fig. 4.** A SFT template for TTS

**Table 3**
A description of the SFT template for TTS

| | | |
|---|---|---|
| Time Nodes | Satisfy the timing constraints | The condition and state are maintained for k_delay time. |
| | Un-satisfy the timing constraints | **- Unsatisfying the Logic condition**<br>The input value entered into TT S at current cycle doesn't satisfy the logic condition regardless of whether the state and condition are satisfied for a certain time.<br>**- Unsatisfying the time condition**<br>The input value entered into TTS at current cycle satisfy the loc,z.ic condition but the state and the condition aren't maintained for elay time. In other words, the state and the condition are maintained for one of the times between t-1 and t-(k_delay-1) |

- During the satisfaction time, the TTS remains in the state and the logic condition has to remain *true*.
- Before the satisfaction time, the TTS enters the state via state transitions or the TTS remains in the state and the logic condition is *false*.

To represent the time nodes in detail, we make a set of software fault trees corresponding to a sequence of the behaviors. ⟨Fig. 5⟩ shows the set of software fault trees when the satisfaction time is *k_delay*.

⟨ Fig. 6 ⟩ and ⟨Fig. 7⟩ are the software fault tree templates for a sequence of behaviors. Each template is used to generate an element of a set of software fault trees. ⟨Fig. 6⟩ is the template for the behaviors during the satisfaction time and it ensures that the TTS is in a specific state at a certain time and satisfies a logic condition. It is used to construct the SFT for the behavior of the TTS from before the current time to before the time to be satisfied (*i.e.,* t-1 ∼ t-(certain time-1)). ⟨Fig. 7⟩ is a template for the behaviors before the satisfaction time and it ensures that a transition to the specific state occurs or remains in that specific state while not satisfying a logic condition. It is used to construct the SFT for the behavior of the TTS at the before a certain time in the current time. (*i.e.,* t-certain time). The elements of each template are the same as the template for the TTS.

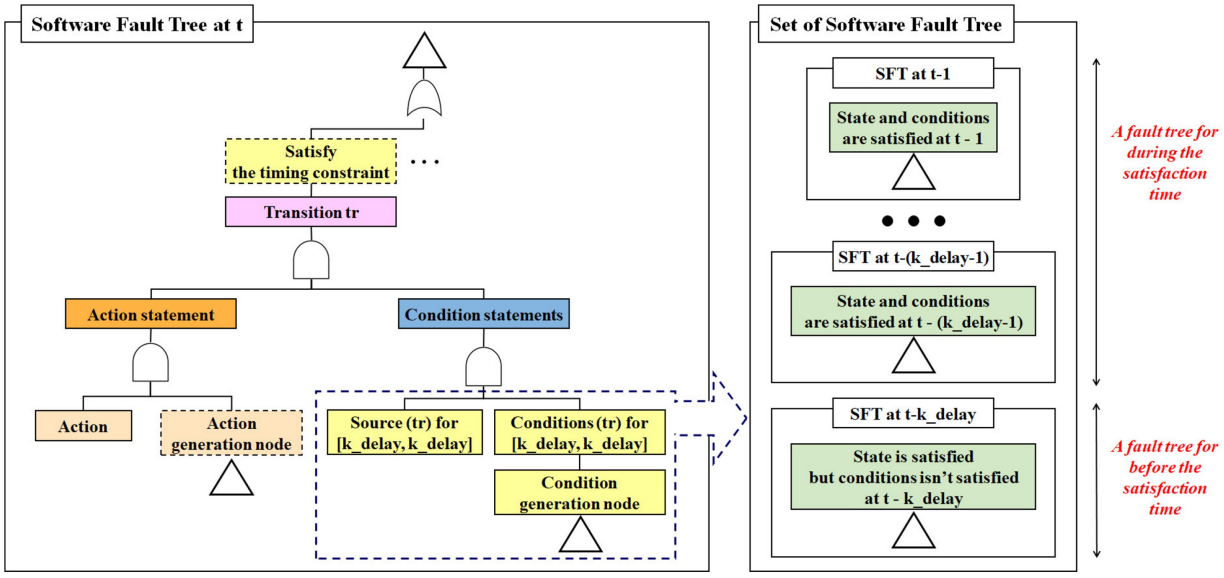



**Fig. 5.** A set of SFT related to the time nodes

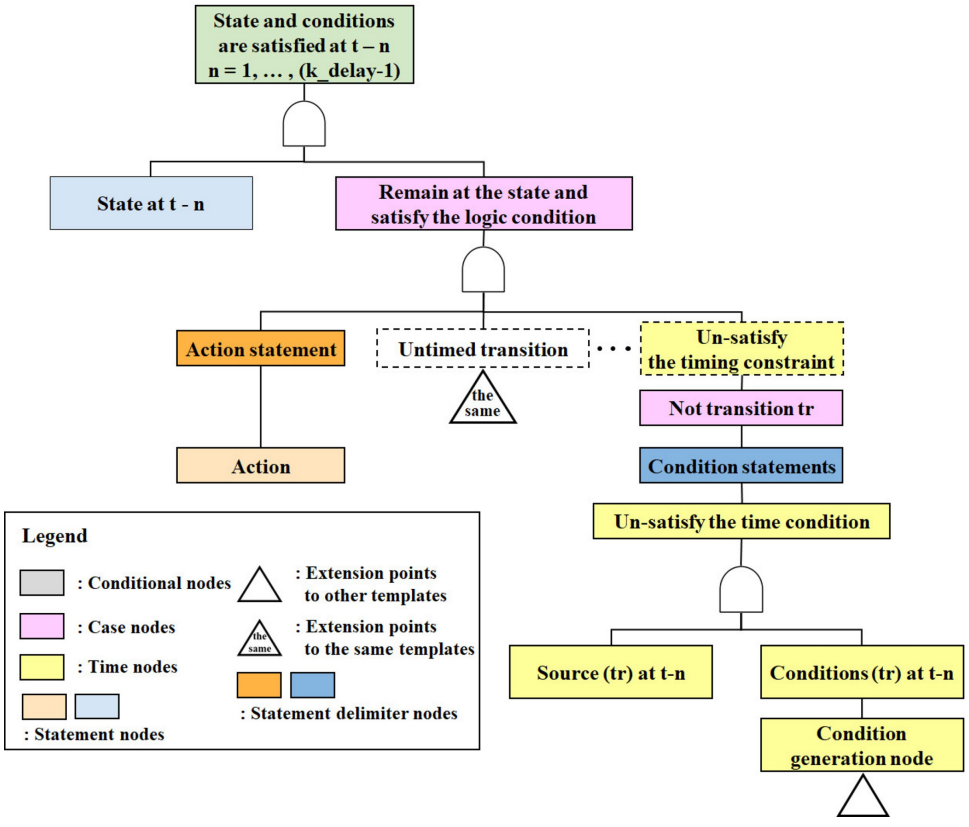



**Fig. 6.** A SFT template for the behaviors during the satisfaction time of the time node

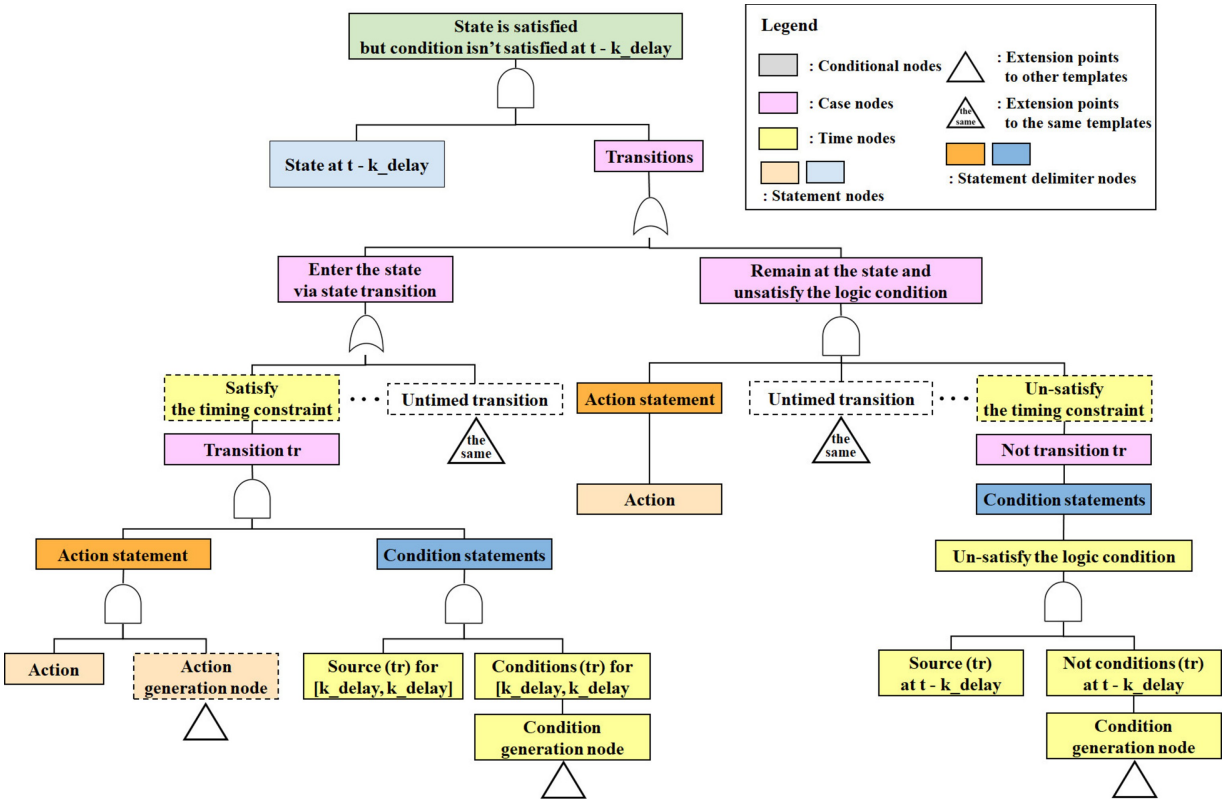



**Fig. 7.** A SFT template for the behaviors before the satisfaction time of the time node

### 3.2. The Software Fault Tree Construction Algorithms

Constructing software fault trees for *NuSCR* requirement specifications involves two algorithms: 1) ***generating a foundation SFT*** : constructs a foundation SFT by exploring all nodes of FODs; 2) ***generating a set of SFTs for multi-cycles*** : constructs a set of SFTs corresponding to the time nodes of the SFTs for multi-cycles. SFT construction firsts constructs a software fault tree by backwardly analyzing possible causes of the output value represented by the root node throughout all connected nodes in FODs. The next step of construction is generating fault trees corresponding to time nodes in the foundation SFTs, it constructs an additional set of fault trees that consider the timing constraints of multi-cycles. ⟨Fig. 8⟩ shows a process of SFT construction algorithm. All SFT construction uses the template which is outlined in the previous section.

#### 3.2.1. The SFT Construction Algorithms for a foundation fault tree

In previous work [16,20], we proposed an SFT construction algorithm corresponding to one single cycle. The software fault tree constructed, however, is too large to perform SFTA, since it uses an explicit search strategy which finds a possible cause by unfolding the symbol value into the concrete value. A symbolic search strategy, which finds a possible cause by changing the symbol value by comparing the symbol value with the output value, needs to overcome the restriction as the cases of formal verification techniques such as symbolic model checking [56].

⟨ Algorithm 2 ⟩ shows the process of generating foundation SFTs. The algorithm recursively expands SFTs from a root node. The expansion terminates when all terminal nodes of the SFT consist of input nodes or states of the NuSCR specification. In lines 2-4, if the variable node is defined as an FSM or TTS, it first unfolds them guided by ⟨Algorithm 1⟩. In line 5, the *setSymbol()* modifies the range of the RHS


**Fig. 8.** A process of the SFT construtions algorithm

**Require:** *node* : The variable node, *output* : The value or the range, *root* : The fault tree node

1: **function** GENERATESOFTWAREFAULTTREE(*node*, *output*, *root*)
2: **if** *node* is *history variable node* or *timed history variable node* **then**
3: $unfoldFSM(node)$ ▷ Algorithm 1
4: **end if**
5: $setSymbol(output, node)$ ▷ Algorithm 3
6: $connectTemplate(node, root)$
7: **Create a fault tree node list** $ftTerminalList$
8: $ftTerminalList \leftarrow findTerminalNodeList(root)$
9: **for all** $terminalNode \in ftTerminalList$ **do**
10: **if** *terminalNode* is *input nodes* or *states* **then**
11: *continue*
12: **else**
13: **Create a variable node** *predecessorNode*
14: **Create a output** *predecessorOutput*
15: $predecessorNode \leftarrow findRelatedNodeInFOD(terminalNode)$
16: $predecessorOutput \leftarrow extractOutput(terminalNode)$
17: **generateSoftwareFaultTree(predecessorNode, predecessorOutput, terminalNode)**
18: **end if**
19: **end for**
20: **end function**

**Algorithm 2.** Generating Software Fault Tree

```
Require: node : The variable node, output : The value or the range
1: function setSymbol(node, output)
2:     Create a RHS of action statement list actionList
3:     actionList ← findActionList(node)
4:     for all action ∈ actionList do
5:         if action.Type is range then
6:             if output.Type is constant then
7:                 if action.min ≤ output.value ≤ action.max then
8:                     action.setValue(output.value)
9:                     markPossibleCause(action)
10:                end if
11:            else if output.Type is range then
12:                if action.min > output.min and action.max > output.max then
13:                    action.setMax(output.max)
14:                    markPossibleCause(action)
15:                else if action.min < output.min and action.max < output.max then
16:                    action.setMin(output.min)
17:                    markPossibleCause(action)
18:                else if action.min < output.min and action.max > output.max then
19:                    action.setMax(output.max)
20:                    action.setMin(output.min)
21:                    markPossibleCause(action)
22:                else if action.min ≥ output.min and action.max ≤ output.max then
23:                    markPossibleCause(action)
24:                end if
25:            end if
26:        else if action.Type is constant then
27:            if output.Type is constant then
28:                if action.value is equal output.value then
29:                    markPossibleCause(action)
30:                end if
31:            else if output.Type is range then
32:                if output.min ≤ action.value ≤ output.max then
33:                    markPossibleCause(action)
34:                end if
35:            end if
36:        end if
37:    end for
38: end function
```

**Algorithm 3.** Setting Symbols

of the action statement and marks the RHS of the action statement, which is an element that can be added to the template. In line 6, the *connectTemplate()* function connects the template corresponding to the variable node to the root. In lines 7-17, if there is a terminal node that has information on the variable node while traversing all the terminal nodes of the tree, it extracts the information about the application of the template related to the node and calls *generateSoftwareFaultTree()* with the extracted information as an argument.

〈 Algorithm 3 〉 is the algorithm for a symbolic search strategy, which is called 'setSymbol' in 〈Algorithm 2〉. It traverses all RHS of the action statements one by one in order to set the range and the marking. If the type of the RHS of the action statement is a constant and its condition is satisfied, then the RHS of the action statement is marked. If the type of the RHS of the action statement is a range and its condition is satisfied, then the RHS of action statement is changed to consist only of the intersection between the output and the RHS of the action statement and then the RHS of the action statement is marked.

### 3.2.2. The SFT Construction Algorithm for Multi-Cycles

〈 Algorithm 4 〉 shows the process of generating a set of SFTs for multi-cycles. In order to generate a set of SFTs for multi-cycles, the timing constraints related nodes must exist in the foundation fault trees generated by the previously explained algorithms. The inputs of the algorithm are as follows: *node* is the timing history variable node associated with the timing constraint; *state* is the source of the transition associated with the timing constraint; *d* is the time that the condition and the state must be maintained in.

〈 Algorithm 4 〉 is very similar to 〈Algorithm 2〉 except for the part that connects the template to the root node. The algorithm generates SFTs as much as the input time and recursively expands each SFT from a root node. The expansion termination condition of each SFT is the same as 〈Algorithm 2〉. The algorithm returns a list of the root nodes of the generated SFT. In lines 4-6, it creates a root node at a certain time, sets the information of the root node, and includes the generated root

**Require:** *node* : The variable node, *state* : The state of the timed history variable node, *d* : The time

1: **function** GENERATESFTFORTIMECYCLES(*node*, *state*, *d*)
2: **Create a fault tree node list** $softwareFaultTreeSet$
3: **for** $i = 1$ to $d$ **do**
4: **Create a fault tree node** *root*
5: *settingRootNode*(*root*, *node*, *i*)
6: $softwareFaultTreeSet.add(root)$
7: **if** $i \neq d$ **then**
8: *connectFirstInterpretationTemplate*(*root*, *node*, *state*, *i*)
9: **else**
10: *connectSecondInterpretationTemplate*(*root*, *node*, *state*, *i*)
11: **end if**
12: **Create a fault tree node list** $ftTerminalList$
13: $ftTerminalList \leftarrow findTerminalNodeList(root)$
14: **for all** $terminalNode \in ftTerminalList$ **do**
15: **if** *terminalNode* is *inputnodes* or *states* **then**
16: *continue*
17: **else**
18: **Create a variable node** *predecessorNode*
19: **Create a output** *predecessorOutput*
20: $predecessorNode \leftarrow findRelatedNodeInFOD(terminalNode)$
21: $predecessorOutput \leftarrow extractOutput(terminalNode)$
22: **generateSoftwareFaultTree(predecessorNode, predecessorOutput, terminalNode)** ▷ Algorithm 2
23: **end if**
24: **end for**
25: **end for**
26: **Return** $softwareFaultTreeSet$
27: **end function**

**Algorithm 4.** Generating Software Fault Tree for Time Cycles

node as an element of the set of software fault trees. In lines 7-10, if the time is not the last time 'd', it connects the template for the first interpretation of the time nodes in the TTS to the root node and if not it connects the template for the second interpretation to the root node. Lines 12-22 are the same as the lines 7-17 of ⟨Algorithm 2⟩.

### 3.3. The Minimal Cut Set Calculation Algorithm

This section explains an algorithm to calculate the MCSs of a SFT. It is based on a dynamic programming strategy and progressive manipulations of the SFT. The algorithm consists of 3 steps: that are (1) converting an SFT into the equivalent set of unique subtrees, (2) calculating MCSs for all sub trees, and (3) expading the MCSs for all subtrees into the MCSs of the SFTs. The SFT has several identical subtrees which are filled with the same content as the SFT template. Thus, to ease the complexity of the problem and eliminate the identical subtrees, step 1 converts the SFT into the equivalent set of unique subtrees, defines the relationships between the subtrees. Step 2 calculates the minimal cut-sets for all subtrees that are generated with the SFT template. Step 3 expands the minimal cut-sets of all subtrees into the minimal cut-sets of the SFT using the law of Boolean algebra [57] with a bottom-up approach.

⟨ Algorithm 5 ⟩ shows the process of converting the SFT into the set of a top node of the subtrees. The algorithm recursively traverses all the nodes of the SFT and identifies the top nodes of the subtrees among all the SFT nodes, stores them in the set of subtrees, and defines the top-to-bottom relationship between the top nodes of the subtrees. The 'compareSFTTemplate()' function checks if two nodes belong to the same subtree, and if they belong to the same subtree, the function recursively calls the algorithm to check the next node, as line 2-4. If not, we check if the node exists in the set of top nodes and if the node exists, it defines the top-down relationship between the top node in the set and the top node, we then terminate the algorithm because all child nodes of the node have already been checked, as line 5-11. In line 12-15, if the node does not exist in the set, it defines the top-down relationship between the node and the top node, adds the node to the set and calls the algorithm recursively.

To obtain the MCSs of the SFT, we calculate the MCSs of the subtree obtained as a prerequisite. We use the MOCUS [58] algorithm that is the oldest deterministic algorithm developed for obtaining MCSs. Since

**Require:** *parentNode* : The top node of subtrees, *node* : The node of the software fault tree, *list* : The set of the top node of subtrees

```
1:  function ConvertingSoftwareFaultTreeintoSetofSubtrees(node, node, list)
2:      for all childNode ∈ node.getChilds() do
3:          if compareSFTTemplate(parentNode, childNode) then
4:              ConvertingSoftwareFaultTreeintoSetofSubtrees(parentNode, childNode, list)
5:          else
6:              for all subtreeNode ∈ list do
7:                  if isEqual(childNode, subtreeNode) then
8:                      subtreeNode.addParentOfSubtree(parentNode)
9:                      parentNode.addChildOfSubtree(subtreeNode)
10:                     return
11:                 end if
12:             end for
13:             childNode.addParentOfSubtree(parentNode)
14:             parentNode.addChildOfSubtree(childNode)
15:             list.add(childNode)
16:             ConvertingSoftwareFaultTreeintoSetofSubtrees(childNode, childNode, list)
17:         end if
18:     end for
19: end function
```

**Algorithm 5.** Converting Software Fault Tree into Set of Subtrees

**Require:** *list* : The set of the top node of subtrees

```
1:  function expandMCSofSubtrees(list)
2:      while list.size > 1 do
3:          for all childSubtree ∈ list do
4:              if list.get(i).hasChildOfSubtree is false then
5:                  for all parentSubtree ∈ childSubtree.getParentOfSubtree() do
6:                      cartesianProduct(parentSubtree, childSubtree)
7:                      lawOfBooleanAlgebra(parentSubtree.getMinimalCutSet())
8:                      parentSubtree.removeChildOfSubtree(childSubtree)
9:                  end for
10:                 list.remove(childSubtree)
11:             end if
12:         end for
13:     end while
14: end function
```

**Algorithm 6.** Expanding MCSs of Subtrees into MCSs of Software Fault Tree

the size of the subtree is the same as the size of one SFT template, the execution speed of the algorithm is not optimal, but there is no problem in finding the MCSs of the subtree. We can expand the MCSs of subtrees into the minimal cut-sets of the SFT for both top-down and bottom-up approaches, because we have defined the top-down relationships between the subtrees in the first step. We use the bottom-up approach because the top-down approach is much more computationally taxing than the bottom-up approach [59].

〈 Algorithm 6 〉 shows the process of expanding the MCSs of the subtrees into the MCSs of the SFT using a bottom-up approach. The algorithm finds a subtree that has no child of subtrees, traverses all the parents of the subtree, and performs a cartesian product the minimal cut-sets of the subtree and the minimal cut-set that containins the subtree of the parent of the sub tree. After the cartesian product, the minimal cut-sets of the parent of the subtree may not meet the MCS condition, so the minimal cut-set condition is adjusted again using the Boolean algebra's idempotent Law and absorption Law. The algorithm repeats this process until only the root node of the SFT remains in the set.

## 4. NuFTA 2.0

"*NuFTA 2.0*" is a supporting tool to do SFTA on NuSCR formal specifications mechanically. When we select an important output node such as a shutdown signal in *NuSRS 2.1*, the *NuFTA 2.0* mechanically generates a SFT and calculates MCSs as depicted in 〈Fig. 9〉, based on the algorithms introduced in Section 3. We constructed the fault trees into sub-trees in module form for improving readability, because the fault tree has too many nodes to view as a single tree. *NuFTA2.0* consists of two components, the software fault tree part and the minimal cut-set part. The software fault tree part shows the SFTs with the zoomed view from the JAVA library *JGraph*. The minimal cut-sets part shows all MCSs for the SFT and saves them in a *.txt* format file to seamlessly accelerate future extensions to this analysis. *NuFTA 2.0* aims

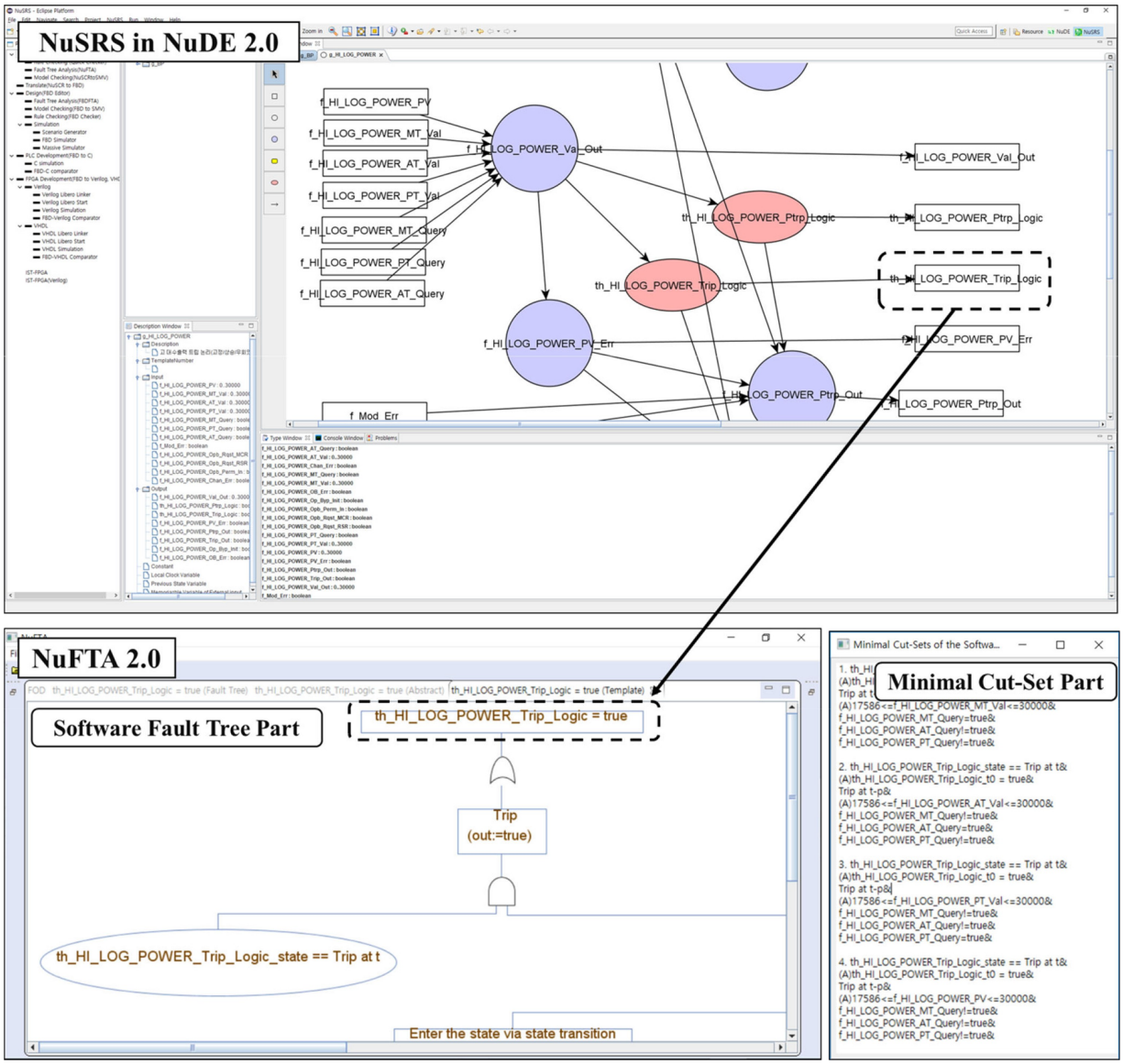



**Fig. 9.** *NuFTA 2.0* : An assistant tool for generating SFT and MCSs from NuSCR specifications

**Table 4**
A summary of SFT construction performance

| | *th_X_Trip_Logic == true* | | *f_X_Trip_Out == true* | |
|---|---|---|---|---|
| | time(s) | # of nodes | time(s) | # of nodes |
| NuFTA 1.0 | 12.477 | 1,609,343 | N/A | N/A |
| NuFTA 2.0 | 0.011 | 123 | 0.144 | 2156 |

X : HI_LOG_POWER

to be a analysis platform for various SFTA, *i.e.,* vertical and horizontal analysis [60].

The *NuFTA 2.0* shows a better performance compared to our previous version [20], besides the *NuFTA 2.0* supports the multi-cycle case of fault trees and MCSs that previous version cannot support. ⟨Table 4⟩ is a summary of the comparison results of two outputs, *th_HI_LOG_POWER_Trip_Logic == true* and *f_HI_LOG_POWER_Trip_Out == true*, in the ⟨Fig. 1⟩. We measured the time taken for SFT construction and the number of SFT nodes constructed. As *NuFTA 1.0* is not able to generate multi-cycle cases for fault trees, we only measured the time and nodes for a single cycle SFT when comparing with *NuFTA 2.0*. *NuFTA 2.0* is faster and more efficient than *NuFTA 1.0* when generating fault trees of the outputs for *th_HI_LOG_POWER_Trip_Logic == true*. The case of *f_HI_LOG_POWER_Trip_Out == true* also showed that the *NuFTA 1.0* cannot afford it within the time limitation while *NuFTA 2.0* did it in a quiet reasoning time and space consumption. The nodes generated by *NuFTA 2.0* are symbolic bundles of concrete values represented in the nodes generated by *NuFTA 1.0*, so the semantics of all nodes created by both tools are the same.

## 5. Case Study

We performed a case study to demonstrate the effectiveness and feasibility of the proposed SFTA technique and the supporting tool *NuFTA 2.0*. Section 5.1 explains the target system software in the case study and section 5.2 shows the analysis results of the proposed SFTA technique, as well as construction results for fault trees and MCSs with multi-cycled requirements. We also introduce an additional usage for the SFTA results in section 5.3.

### 5.1. Target System Software

We used a *g_HI_LOG_POWER* module which is the initial version of a BP software module for the APR-1400 RPS operating in Korea[24]. The RPS is a safety-critical system whose main purpose is to protect reactors during unexpected behavior in the NPP. The software in the RPS is crucial to the safety in that its malfunction may result in irreversible consequences. The *g_HI_LOG_POWER* module is one of 18 modules in the BP software. It is a '*fixed set point rising trip*' logic that fires a trip signal when the system faces an emergency threatening the safety of the system. The trip signal is triggered when the state variable *f_HI_LOG_POWER_PV* has been maintained above a set point for a period of time. ⟨Fig. 10⟩ shows the value of the *f_HI_LOG_POWER_PV* variable over time that occurs the trip signal according to the cycle time. ⟨Fig. 1⟩ shows the FOD and TTS for the *HI_LOG_POWER* logic which is simplified version with eliminating pre-trip algorithms.

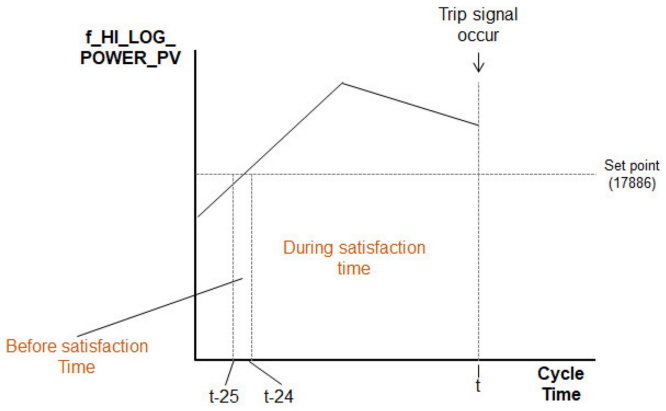



**Fig. 10.** A multi-cycle example of a value changing with time for the *g_HI_LOG_POWER* logic

### 5.2. SFT Construction and MCS Analysis

We constructed fault trees and MCSs that are top event conditions of *f_HI_LOG _POWER_Trip_Out == true* from the *g_HI_LOG_POWER* logic. The system scan cycle time applied is 20ms to check the timing constraints via the TTS node in this paper. *k_HI_LOG_POWER_Trip_Delay* is divided by the scan cycle time for checking time flows. ⟨Fig. 11⟩ is an example of the generated SFTs. We can obtain 512 kinds of MCSs from the results, and these MCSs contain either single cycles or multi-cycles. The structure of these MCS is ((MCS1) ‖ (MCS2) ‖ · · · ‖ (MCS512)), that are an OR combination between a set of conditions which satisfy *f_HI_LOG_POWER_Trip_Out == true*. Each MCS consists of a combination of variables and their specific values. ⟨Table 5⟩ shows some examples of the MCSs among the 512 MCSs. Case 1 ∼ 3 appears different condition for satisfying the top-event respectively.

Case 1 MCS appears the states about "*the module fires a trip signal (f_HI_LOG _POWER _Trip_Out == true) because the process variable value is not a valid range.*" Case 3 MCS contains a timing constraint *Waiting for [480, 480]* which needs to be analyzed multi-cycled time. It means that the operation value *f_HI_LOG_POWER_PV* is greater than the trip-set-point for a *480* time to satisfy the top-event. The proposed SFTA technique can construct fault trees and calculate MCSs about these multi-cycled time cases and the multi-cycled MCS is divided into the 'during' and 'before' the satisfaction time. ⟨Fig. 10⟩ shows an example of during and before the satisfaction time in the multi-cycle MCS for '*Waiting for [480, 480]*,' ⟨Table 6⟩ shows MCSs about multi-cycled timing requirements about during and before the satisfaction time in the case study. The MCSs about during the satisfaction time affects at the *n* and *n+1* cycle times and MCSs about before the satisfaction time affects at the *t-24* and *t-25* cycle times.

⟨ Table 7 ⟩ shows a summary of the software fault tree analysis by *NuFTA 2.0*. We found a logic error and hazardous states in the NuSCR requirement specification in the *g_HI_LOG_POWER* logic. Case 2 MCS reveals a condition for firing the trip signal, however, if we check the three range conditions ($0 \le f\_HI\_LOG\_POWER\_Val\_Out \le 17885$, $29401 \le f\_HI\_LOG\_POWER_Val\_Out \le 30000$, and $0 \le$ f_HI_LOG_POWER _ Val_Out $\le$ 599) to check the range of the process variable value, we can see that there is no intersection. This condition state

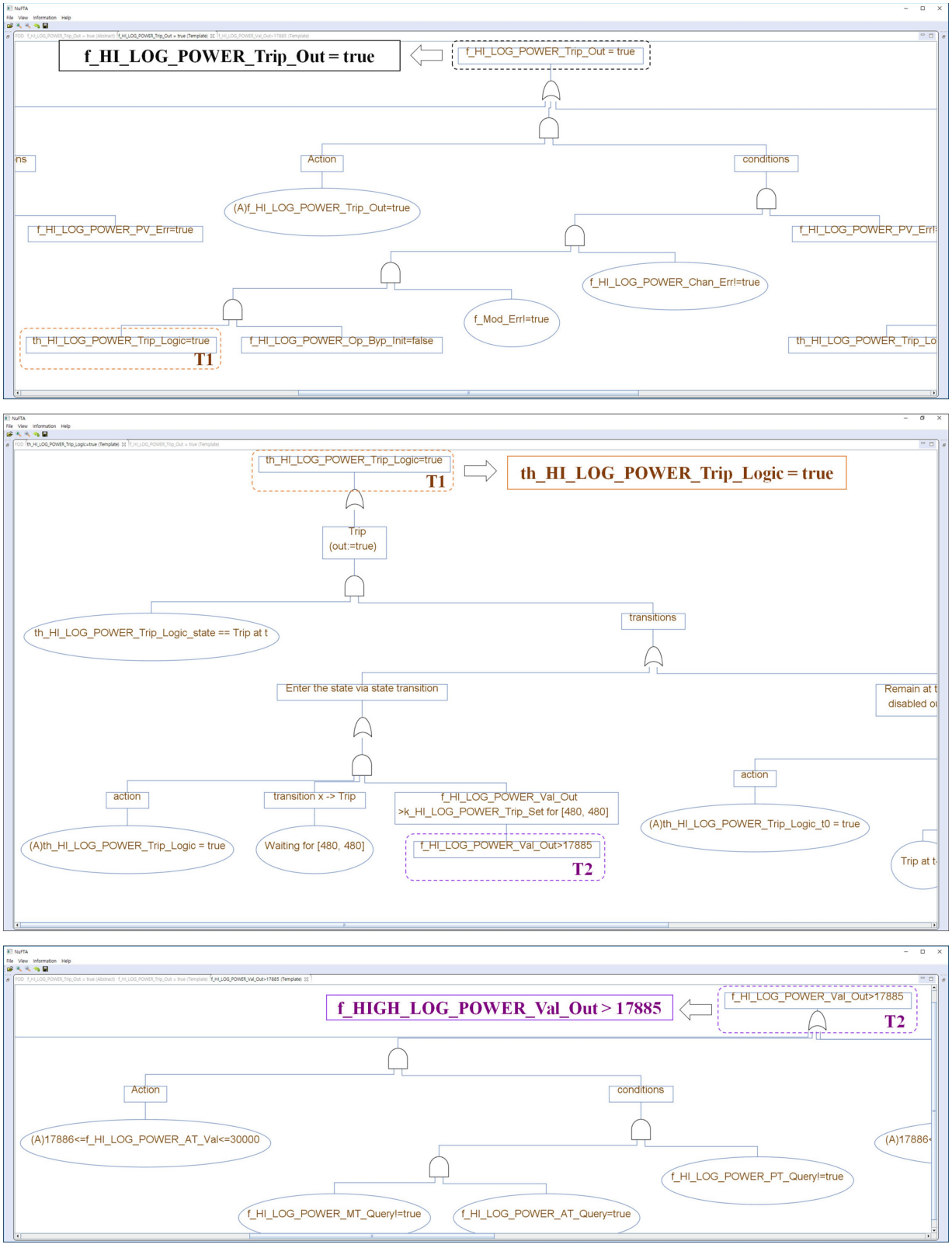



**Fig. 11.** A Software Fault Tree with the Top Node (*f_HI_LOG_POWER_Trip_Out == true*)

**Table 5**
Three examples of MCSs from the fault trees generated by the case study

| Element | MCS case 1 | MCS case 2 | MCS case 3 |
|---|---|---|---|
| (A) f_X_Trip_Out | true | true | true |
| & (A) f_X_PV_Err | true | true | false |
| & (A) th_X_Trip_Logic | false | false | true |
| & th_X_Trip_Logic's state at t | Waiting | Normal | Trip |
| & th_X_Trip_Logic's state at t-1 | Waiting | Waiting | Waiting for [480, 480] |
| & (A) f_X_Val_Out | 17886 ≤ v ≤ 30000 & 29401 ≤ v ≤ 30000 & 600 ≤ v ≤ 30000 | 0 ≤ v ≤ 17885 & 29401 ≤ v ≤ 30000 & 0 ≤ v ≤ 599 | 17886 ≤ v ≤ 30000 & 0 ≤ v ≤ 29400 & 600 ≤ v ≤ 30000 |
| & f_X_PV | 17886 ≤ v ≤ 30000 & 29401 ≤ v ≤ 30000 & 600 ≤ v ≤ 30000 | 0 ≤ v ≤ 17885 & 29401 ≤ v ≤ 30000 & 0 ≤ v ≤ 599 | 17886 ≤ v ≤ 30000 & 0 ≤ v ≤ 29400 & 600 ≤ v ≤ 30000 |
| & f_X_AT_Val | N/A | N/A | N/A |
| & f_X_PT_Val | N/A | N/A | N/A |
| & f_X_MT_Val | N/A | N/A | N/A |
| & f_X_AT_Query | false | false | false |
| & f_X_PT_Query | false | false | false |
| & f_X_MT_Query | false | false | false |
| & (A) f_X_Op_Byp_Init | false | false | false |
| & f_X_AT_Opb_Rqst_MSR | false | false | N/A |
| & f_X_AT_Opb_Rqst_RSR | false | false | N/A |
| & f_X_AT_Opb_Perm_In | N/A | N/A | false |
| & f_Mod_Err | N/A | N/A | false |
| & f_X_Chan_Err | N/A | N/A | false |
| X : HI_LOG_POWER | | | |

**Table 6**
MCSs about 'during' and 'before' the satisfaction time for multi-cycled timing constraints

| Element | MCSs for multi-cycle by '*Waiting for [480, 480]*' | | | | | |
|---|---|---|---|---|---|---|
| | '*During the satisfaction time*' | | '*Before the satisfaction time*' | | | |
| | Case 1 | Case 2 | Case 1 | Case 2 | Case 3 | Case 4 |
| (A) th_X_Trip_Logic | false | false | false | false | false | false |
| & th_X_Trip_Logic's state at t-n / t-24 | Waiting | Waiting | Waiting | Waiting | Waiting | Waiting |
| & th_X_Trip_Logic's state at t-(n+1) / t-25 | Waiting | Waiting | Normal | Normal | Waiting | Waiting |
| & (A) f_X_Val_Out | 17886 ≤ v ≤ 30000 | 17886 ≤ v ≤ 30000 | 17886 ≤ v ≤ 30000 | 17886 ≤ v ≤ 30000 | 17886 ≤ v ≤ 30000 & 0 ≤ v ≤ 17885 | 17886 ≤ v ≤ 30000 & 0 ≤ v ≤ 17885 |
| & f_X_PV | 17886 ≤ v ≤ 30000 | N/A | 17886 ≤ v ≤ 30000 | N/A | 17886 ≤ v ≤ 30000 & 0 ≤ v ≤ 17885 | N/A |
| & f_X_AT_Val | N/A | 17886 ≤ v ≤ 30000 | N/A | 17886 ≤ v ≤ 30000 | N/A | 17886 ≤ v ≤ 30000 & 0 ≤ v ≤ 17885 |
| & f_X_PT_Val | N/A | N/A | N/A | N/A | N/A | N/A |
| & f_X_MT_Val | N/A | N/A | N/A | N/A | N/A | N/A |
| & f_X_AT_Query | false | true | false | true | false | true |
| & f_X_PT_Query | false | false | false | false | false | fals |
| & f_X_MT_Query | false | false | false | false | false | false |
| X : HI_LOG_POWER | | | | | | |

**Table 7**
An analysis result of SFTA by *NuFTA 2.0*

| MCSs (in 〈 Table 5 〉) | | Analysis Description | Result |
|---|---|---|---|
| MCS 1 | | A case of handling error value | |
| MCS 2 | | Erroneously specification | Logic error |
| MCS 3 | Case 1 (during) & Case 1 (before) | A case regarding the timing constraint for tirp signal occurs | |
| | Case 2 (during, before) | Contrary condition to SRS | Hazardous state |
| | Case 3, 4 (before) | Conflict condition, but intended specification | |

means that the NuSCR specification associated with the case 2 MCS, contains erroneously specified.

As shown in 〈Table 6〉, the MCSs fire a trip signal if only the case where the process variable value is greater than the trip set point (17886 ≤ f_HI_LOG_POWER_Val _Out ≤ 30000) in state '*Normal*' at t-25, it transits to the '*Waiting*' state at t-24 and then the conditions (17886 ≤ *f _HI_LOG_POWER_Val_Out* ≤ 30000) lasts for a time delay (*24 cycle*) in the '*Waiting*' state. In the informal software requirement specification (SRS), the timing constraint is described as follows: *"If the process variable value is greater than the trip set-point and then this state lasts for a time delay, the module fires a trip signal."* It can be confirmed that there is no contradiction between the MCS and the informal specifications, which is no corresponding part of the hazard item/logical error regarding the timing constraint.

We found a contrary condition between the NuSCR specification from the case 3 MCS with multi-cycled cases about case 2 in during and before the satisfaction time. The constraints of the automatic testing are described as follows: *"The automatic test is run on the test scan to avoid*

*affecting the actual operation."* According to the informal SRS, the automatic testing value (*f_HI_LOG_POWER_AT_Val*) and the actual operation value (*f_HI_LOG_POWER_PV*) must be independent in firing a trip signal. However, case 2 MCSs of during and before the satisfaction time are contrary to the requirement specification. We can construct software fault trees and MCSs automatically from *NuFTA 2.0* and we are also able to identify several hazardous state and logic errors of the NuSCR specification by the SFTA in this paper.

### 5.3. Various Uses of the SFTA Results

We could get various cases of MCSs from the NuSCR requirement specification using *NuFTA 2.0*. This section introduces the various uses of the SFTA results for software analysis. First, the MCSs can be used to generate simulation/testing scenarios for a later phase of the software development. Simulation scenarios commonly consist of input combinations according to the time step. MCSs for multi-cycled conditions illustrates these combinations well by a time flow. 〈 Table 8 〉 is an example of simulation scenarios generated from the case 3 MCSs in 〈Table 5〉 and its multi-cycled cases. It consists of 26 cycles for simulating the functionality of the specification about 'trip signal occur when a state variable *f_HI_LOG_POWER_PV* has been maintained above a set point for a period of time.' The value 17886 is a set point which is marked in 〈Fig. 10〉 and the input value of the *f_HI_LOG_POWER_PV* variable consists of maintaining a higher value than 17886 in the scenario.

〈 Fig. 10 〉 is one example of a value changing that satisfies the simulation scenario generated. The scenario contains state/variable changes for representing sequence of behavior of the software-based controllers in time flows to simulate. According to the MCSs, the scenario also can contain simple event behavior. If simulation-based testing is proceed using scenarios from SFTA of NuSCR, the simulation can covers functionalities of the specification including timing-constrained continuous behaviors compared with other test case generation methods using fault trees [48,50]. Scenarios from SFTA of NuSCR can be used as simulation input of NuSCR itself or after development artifacts such as FBD design model, Verilog, or VHDL[31,61].

Another case of using the MCSs is for identifying software contributable hazards by combining guide words. Guide words are a predefined set for identifying deviations from software or system elements, MCSs can be used as a specific candidate for applying guide words to analyze software contributable hazards. For example, analyst identifies which deviation occurs when the state of guide phrases under a certain condition of the MCSs. 〈Table 9〉 shows some results of the software hazard anlayiss by applying guide phrases about 'functionality' and 'safety' of the [8]. MCS 1 and 3 in 〈Table 5〉 can support deviation analysis by providing a certain set of states or the condition of the software for software hazard analysis.

**Table 8**
An example of simulation scenario from MCSs of the "*NuFTA 2.0*"

| Cycle | | 0 | 1 | ... | 24 | 25 |
|---|---|---|---|---|---|---|
| Input | f_X_PV | 15000 | 17886 | 17886 ≤ v ≤ 30000 | 19342 | 19632 |
| | f_X_AT_Query | false | false | false | false | false |
| | f_X_MT_Query | false | false | false | false | false |
| | f_X_PT_Query | false | false | false | false | false |
| | f_X_Mod_Err | false | false | false | false | false |
| | f_X_Chan_Err | false | false | false | false | false |
| Expected output | f_X_Trip_Out | false | false | false | false | true |

X : HI_LOG_POWER

**Table 9**
An example of software hazard analysis using guide phrases

| Item | Guide phrases | Deviation (combining MCS & Guide phrases) | Hazard | Hazardous |
|---|---|---|---|---|
| RPS BP software | Function executes when trigger conditions are not satisfied | *g_HI_LOG_POWER* module does not fire the trip signal under condition of case 1-3 MCSs | MCS 1: A system does not receive a trip signal, when a process variable is in an valid error range | O |
| | | | MCS 2: Requirement logic error | X |
| | | | MCS 3: A system does not receive a trip signal, when the system is in a critical state | O |
| | Function executes when trigger conditions are not satisfied | *g_HI_LOG_POWER* module fire the trip signal under condition *f_HI_LOG_POWER_Trip_Out == false* | A system receive trip signal when the system is in normal state | O |
| | Software fails to recognize hazardous reactor state | *g_HI_LOG_POWER* module does not recognize the error value under condition of case 1, 3 MCSs | A system does not receive a trip signal, when a process variable is in an error or hazardous value | O |

## 6. Conclusion and future work

For the safety analysis of safety-critical software, this paper presented an SFTA technique for NuSCR formal requirements specification. We introduced the software fault tree templates for the construction of SFTs from each nodes of the NuSCR and redefined algorithms for fault tree construction and minimal cut-set analysis. The proposed algorithm and template can cover timing constraints which prevous versions were unable to handle. We also proposed a supporting tool, "*NuFTA 2.0*," for generating SFTs and MCSs automatically from NuSCR software requirements specification. We performed the case study using the NuSCR specification for the APR-1400 RPS BP software to demonstrate the feasibility of *NuFTA 2.0* in the software hazard analysis of safety system. The case study showed that *NuFTA 2.0* can construct fault trees and MCSs containing multi-cycled timing constraints. We also introduced various uses of the SFTA results that are scenario generation for simulation and software hazard analysis. We have a plan to make a test scenario from the fault trees with MCSs mechanically/automatically. We are also planning to perform software hazard analysis semi-automatically by combining other analysis techniques.

## Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## CRediT authorship contribution statement

**Sejin Jung:** Methodology, Validation, Visualization, Writing - original draft, Writing - review & editing. **Junbeom Yoo:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing. **Young-Jun Lee:** Resources, Validation, Writing - review & editing.

## Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A1B03030065) and Next-Generation Information Computing Development Program through the National Research Foundation (NRF) of Korea funded by the Ministry of Science, ICT (NRF-2017M3C4A7066479).

## References

[1] Son J, Kim Y, Jeong K, Lee D-A, Yoo J. NuFTA 2.0: New Templates and an Automatic Generator of Fault Tree for NuSCR. Transactions of the Korean Nuclear Society Autumn, Meeting Gyeongju, Korea. 2016.
[2] Criteria for Use of Computers in Safety Systems of Nuclear Powr Plants(RG 1.152). Tech. Rep.. U.S. Nuclear Regulatory Commision (NRC); 2004.
[3] Criteria for Safety Systems(RG 1.153). Tech. Rep.. U.S. Nuclear Regulatory Commision (NRC); 1996.
[4] IEEE Standard Criteria for Safety Systems for Nuclear Power Generating Stations (IEEE 603). Tech. Rep.. Institute of Electrical and Electronics Engineers (IEEE); 2009.
[5] IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations (IEEE 7-4.3.2). Tech. Rep.. Institute of Electrical and Electronics Engineers (IEEE); 2010.
[6] Functional safety of electrical, electronic and programmable electronic (E/E/PE) safety-related systems (IEC 61508). Tech. Rep.. International Electrotechnical Commission (IEC); 2010.
[7] IEEE Standard for Software Safety Plans (IEEE 1228). Tech. Rep.. Institute of Electrical and Electronics Engineers (IEEE); 1994.
[8] Software Safety Hazard Analysis (NUREG/CR-6430). Tech. Rep.. United States Nuclear Regulatory Commission (NRC); 1995.
[9] Developing Software Lift Cycle Processes for Digital Computer Software Used in Safety Systems of Nuclear Power Plants(RG 1.173). Tech. Rep.. U.S. Nuclear Regulatory Commision (NRC); 1997.
[10] IEEE Standard for Developing Software Life Cycle Processes (IEEE 1074). Tech. Rep.. Institute of Electrical and Electronics Engineers (IEEE); 1995.
[11] Lee J-S, Lindner A, Choi J-G, Miedl H, Kwon K-C. Software safety lifecycles and the methods of a programmable electronic safety system for a nuclear power plant. International Conference on Computer Safety, Reliability, and Security. Springer; 2006. p. 85–98.
[12] Park G-Y, Kim DH, Lee DY. Software FMEA analysis for safety-related application software. Annals of Nuclear Energy 2014;70:96–102.
[13] Leveson N.G.. SafeWare: system safety and computers. 1995. ????
[14] Ericson CA. Hazard Analysis Techniques for System Safety. John Wiley & Sons; 2015.
[15] Oh Y, Yoo J, Cha S, Son H. Software safety analysis of function block diagrams using fault trees. Reliability Engineering & System Safety 2005;88(3):215–28.
[16] Kim T, Yoo J, Cha S. A Synthesis Method of Software Fault Tree from NuSCR Formal Specification using Templates. Journal of Korean Institute of Information Scientists and Engineers - Software and Applications (in Korean) 2005;32(12):1178–91.
[17] Park G-Y, Koh KY, Jee E, Seong PH, Kwon K-C, Lee DH. Fault tree analysis of KNICS RPS software. Nuclear Engineering and Technology 2008;40(5):397–408.
[18] Vyas P, Mittal R. Eliciting additional safety requirements from use cases using SFTA. 2012 1st International Conference on Recent Advances in Information Technology (RAIT). IEEE; 2012. p. 163–9.
[19] Dickerson CE, Roslan R, Ji S. A formal transformation method for automated fault tree generation from a uml activity model. IEEE Transactions on Reliability 2018;67(3):1219–36.
[20] Yoon S, Jo J, Yoo J. A domain-specific safety analysis for digital nuclear plant protection systems. Secure Software Integration & Reliability Improvement Companion (SSIRI-C), 2011 5th International Conference on. IEEE; 2011. p. 68–75.
[21] Yoo J, Kim T, Cha S, soo Lee J, Son H. A formal software requirements specification method for digital nuclear plant protection systems. Journal of Systems and Software 2005;74(1):73–83.
[22] Choi J-G, Lee D-Y. Development of RPS trip logic based on PLD technology. Nuclear Engineering and Technology 2012;44(6):697–708.
[23] Kechris A. Classical descriptive set theory. 156. Springer Science & Business Media; 2012.
[24] Korea Atomic Energy Rearch Institute. SRS for Reactor Protection System. Tech. Rep.. 2005. KNICS-RPS-SRS121 Rev.00
[25] Fault Tree Analysis (FTA) (IEC 61025). Tech. Rep.. International Electrotechnical Commission (IEC); 2006.
[26] Leveson NG, Cha SS, Shimeall TJ. Safety verification of ada programs using software fault trees. IEEE software 1991;8(4):48–59.
[27] Kaiser B, Liggesmeyer P, Mäckel O. A new component concept for fault trees. Proceedings of the 8th Australian workshop on Safety critical systems and software-Volume 33. Australian Computer Society, Inc.; 2003. p. 37–46.
[28] Vyas P, Mittal R. The applications of SFTA and SFMEA approaches during software development process: an analytical review. International Journal of Critical Computer-Based Systems 2015;6(1):29–49.
[29] ISO 26262, road vehicles – functional safety. Tech. Rep.. International Organization for Standardization (ISO); 2011.
[30] Heninger KL. Specifying software requirements for complex systems: New techniques and their application. IEEE Transactions on Software Engineering 1980(1):2–13.
[31] Kim E-S, Lee D-A, Jung S, Yoo J, Choi J-G, Lee J-S. NuDE 2.0: A Formal Method-based Software Development, Verification and Safety Analysis Environment for Digital I&Cs in NPPs. Journal of Computing Science and Engineering 2017;11(1):9–23.
[32] Yoo J, Kim E-S, Lee D-A, Choi J-G, Lee YJ, Lee J-S. NuDE 2.0: A Model-based Software Development Environment for the PLC & FPGA based Digital Systems in Nuclear Power Plants. International Symposium of Integrated Circuit (ISIC). 2014.
[33] Parnas DL, Madey J. Functional documentation for computer systems engineering: version 2. McMaster University, Faculty of Engineering, Communications Research Laboratory; 1991.
[34] Henzinger TA, Manna Z, Pnueli A. Timed transition systems. Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems). Springer; 1991. p. 226–51.
[35] Tiwari S, Rathore SS, Gupta S, Gogate V, Gupta A. Analysis of use case requirements using SFTA and SFMEA techniques. 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems. IEEE; 2012. p. 29–38.
[36] Yoo J, Cha S, Son H. Automatic generation of goal-tree from statecharts requirements specification. TRANSACTIONS-AMERICAN NUCLEAR SOCIETY 2003:37–8.
[37] Ratan V, Partridge K, Reese J, Leveson NG. Safety analysis tools for requirements specifications. Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on. IEEE; 1996. p. 149–60.
[38] Programmable Controllers - Part3: Programming languages (IEC 61131-3). Tech. Rep.. International Electrotechnical Commission (IEC); 2013.
[39] Lauer C, German R, Pollmer J. Fault tree synthesis from UML models for reliability analysis at early design stages. ACM SIGSOFT Software Engineering Notes 2011;36(1):1–8.
[40] Oveisi S, Ravanmehr R. Sfta-based approach for safety/reliability analysis of operational use-cases in cyber-physical systems. Journal of Computing and Information Science in Engineering 2017;17(3).
[41] Li L, Lu M, Gu T. A reuse-oriented auxiliary construction method for software fault tree and tool implementation. 2014 10th International Conference on Reliability, Maintainability and Safety (ICRMS). 2014. p. 451–6.
[42] Kabir S. An overview of fault tree analysis and its application in model based dependability analysis. Expert Systems with Applications 2017;77:114–35.
[43] Mhenni F, Choley J-Y, Nguyen N. An integrated design methodology for safety critical systems. 2016 Annual IEEE Systems Conference (SysCon). IEEE; 2016.

p. 1–6.
[44] Yakymets N, Jaber H, Lanusse A. Model-based system engineering for fault tree generation and analysis. the 1th International Conference on Model-Driven Engineering and Software Development. 2013.
[45] Höfig K, Zeller M, Heilmann R. Alfred: a methodology to enable component fault trees for layered architectures. 2015 41st Euromicro Conference on Software Engineering and Advanced Applications. IEEE; 2015. p. 167–76.
[46] Baklouti A, Nguyen N, Mhenni F, Choley J-Y, Mlika A. Dynamic fault tree generation for safety-critical systems within a systems engineering approach. IEEE Systems Journal 2019;14(1).
[47] Alshboul B, Petriu DC. Automatic derivation of fault tree models from sysml models for safety analysis. Journal of Software Engineering and Applications 2018;11(5):204–22.
[48] Tiwari S, Gupta A. An approach to generate safety validation test cases from uml activity diagram. 2013 20th Asia-Pacific Software Engineering Conference (APSEC). 1. IEEE; 2013. p. 189–98.
[49] Li HW, Ren Y, Wang LN. Research on software testing technology based on fault tree analysis. Procedia Computer Science 2019;154:754–8.
[50] Philip G, Dsouza M, Abidha V. Model based safety analysis: Automatic generation of safety validation test cases. 2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC). IEEE; 2017. p. 1–10.
[51] Yadav HB, Yadav DK. Early software reliability analysis using reliability relevant software metrics. International Journal of System Assurance Engineering and Management 2017;8(4):2097–108.
[52] Sinha S, Goyal NK, Mall R. Early prediction of reliability and availability of combined hardware-software systems based on functional failures. Journal of Systems Architecture 2019;92:23–38.
[53] Ali SR. Software reliability analysis. Next Generation and Advanced Network Reliability Analysis. Springer; 2019. p. 59–104.
[54] Mealy GH. A method for synthesizing sequential circuits. Bell Labs Technical Journal 1955;34(5):1045–79.
[55] Moore EF. Gedanken-experiments on sequential machines. Automata studies 1956;34:129–53.
[56] McMillan KL. Symbolic model checking. Symbolic Model Checking. Springer; 1993. p. 25–60.
[57] Sikorski R, Mathématicien P. Boolean algebras. 2. Springer; 1969.
[58] Fussell J, Henry E, Marshall N. MOCUS: a computer program to obtain minimal sets from fault trees. Tech. Rep.. Aerojet Nuclear Co., Idaho Falls, Idaho (USA); 1974.
[59] Garribba S, Garribba P, Naldi F, Reina G, Volta G. Efficient construction of minimal cut sets from fault trees. IEEE Transactions on reliability 1977;26(2):88–94.
[60] Cha S, Yoo J. A safety-focused verification using software fault trees. Future Generation Computer Systems 2012;28(8):1272–82.
[61] Kim J, Kim E-S, Yoo J, Lee YJ, Choi J-G. An integrated software testing framework for FPGA-based controllers in nuclear power plants. Nuclear Engineering and Technology 2016;48(2):470–81.