SMV model-based safety analysis of software requirements

Kwang Yong Koh, Poong Hyun Seong *

Department of Nuclear and Quantum Engineering, Korea Advanced Institute of Science and Technology, 373-1, Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea

1. Introduction

Software control in safety-critical systems such as aerospace, military, nuclear power plant, and medical applications has become more prevalent in recent years. When software is used as a control agent in these systems, safety becomes a paramount concern. In the worst case, software malfunctions can result in serious and unacceptable consequences such as death, injury, or environmental damage [1]. In order to ensure that safety-critical software is adequately safe, attention must be paid to safety concerns from early phases of the development of safety-critical software [2]. Among all the phases in software development, the requirements analysis phase is generally considered to play the most critical role in determining the overall software safety and quality, because defective requirement specifications may result in errors, which propagate to subsequent phases of software development, and mistakes made during the requirements analysis phase can easily introduce faults that subsequently lead to accidents [3].

The safety-critical software process is composed of a development process, a verification and validation (V&V) process, and a safety analysis process. The software requirements specification is an artifact of the requirements phase in the development process. The V&V process in the requirements phase checks the correctness of the software requirements specifications, and the safety analysis process analyzes the safety-related properties in detail [4].

A fault tree analysis (FTA) is one of the most frequently applied safety analysis techniques when developing safety-critical industrial systems such as software-based emergency shutdown systems of nuclear power plants and has been used for safety analysis of software requirements in the nuclear industry. However, the conventional method for safety analysis of software requirements has several problems in terms of correctness and efficiency; the fault tree generated from natural language specifications may contain flaws or errors while the manual work of safety verification is very labor-intensive and time-consuming. In this paper, we propose a new approach to resolve problems of the conventional method; we generate a fault tree from a symbolic model verifier (SMV) model, not from natural language specifications, and verify safety properties automatically, not manually, by a model checker SMV. To demonstrate the feasibility of this approach, we applied it to shutdown system 2 (SDS2) of Wolsong nuclear power plant (NPP). In spite of subtle ambiguities present in the approach, the results of this case study demonstrate its overall feasibility and effectiveness.
can cover all information of the fault tree from natural language specifications, and safety properties obtained by the FTA are automatically verified and the results are analyzed.

The remainder of this paper is organized as follows. Section 2 briefly introduces a model checker SMV, the FTA, and the synthesis procedure of a fault tree from the SMV model. In Section 3, we describe the safety verification and analysis method with SMV and fault tree. Section 4 presents a case study performed on Wolsong SDS2 and the related results. A summary and conclusion are provided in Section 5.

2. Fault tree synthesis from SMV model

2.1. Symbolic model verifier (SMV) system

Model checking is the process of exploring a finite state space to determine whether or not a property holds. A specification is translated to an input for the model checker, possibly with some simplifications. The input and the property that is being tested are then converted to the internal representation of the model checker. The representations are passed to the model checking algorithm. The result is either a claim that the property is true or else a counterexample showing that the property is false. The result can be analyzed by the software engineer to refine the model of the specification, the property, or even the specification itself. This iterative process is inherent in our work.

SMV [7,8] is a representative model checker for finite state systems against specifications written in temporal logic such as computation tree logic (CTL) [9,10], and uses ordered binary decision diagrams (OBDD) to represent the state set and transition relations. It can efficiently use system memory and solve a state explosion problem at some level, and has its own language to describe system models and specifications. The language provides a means for the symbolic description of state transition relations in finite state systems. A SMV model has a modular structure consisting of a main module for global specifications and an arbitrary number of further modules for describing the model components. Each module is declared by a keyword MODULE in combination with the module name. Three sections denoted by the keywords VAR, DEFINE, and ASSIGN follow this declaration, and the main module additionally contains a section denoted by SPEC. These sections have the following meaning.

VAR: This section declares all variables required for the module specification, in which the variables can be either of Boolean type or can be defined on symbolic or finite numeric sets. For the main module, the VAR section declares instances of the other module types including the input variables.

DEFINE: The DEFINE section can be used to associate symbols with a value (true or false) of a logical expression. This construction is often helpful to generate more concise SMV programs.

ASSIGN: This section contains the evaluation of new values for the declared variables. The assignment ‘next(variable) := expression’ computes a new value of the variable on the basis of evaluation of an expression. The latter can involve the current values of the variable and all other variables, as well as conditional expressions. The ASSIGN section can include the statement ‘init(variable) := value’ to initialize a variable; if a variable is not

Fig. 1. Conventional method for safety analysis of software requirements.

Fig. 2. Proposed method for safety analysis of software requirements.
initialized, all values for which the variable is defined are possible initial values.

SPEC: The main module must include a SPEC section that defines the system property to be checked as CTL formulae.

2.2. Fault tree analysis

A FTA [11] is primarily a means for analyzing causes of hazards, not identifying hazards. FTA is a deductive safety analysis technique and a top-down approach [5,12] whose input consists of knowledge of the system’s functions as well as its failure modes and their effects. The results of the analysis are a set of combinations of component failures that can result in a specific malfunction. The approach is graphical, constructing fault trees using standardized symbols.

A fault tree has a root that represents a total or catastrophic failure of a system. The top event in the tree must have been foreseen and thus identified first by other techniques. For each catastrophic failure, the system is analyzed to reveal the possible causes of this failure. FTA uses Boolean logic to describe the combinations of individual faults that can constitute a hazardous event. Each level in the tree lists the more basic events that are necessary and sufficient to cause the problem shown in the level above it. Each sub-tree may be regarded as a fault tree for a component constituting a system of its own. The extent of the analysis, i.e., which components are considered basic, depends on the abstraction level chosen. Once the tree is constructed, it can be written as a Boolean expression and simplified to show the specific combinations of identified basic events sufficient to cause the undesired top event. A fault tree is not a model of all possible causes for system failure; rather, given a particular failure, it reveals the possible combinations of components failures that may lead to this failure.

2.3. Synthesis procedure of fault tree

In this section, we present a procedure for synthesis of a fault tree from the SMV model, which is a given system modeled by SMV input language, with a simple example. The procedure is as follows:

Step 1. Identify the hazardous software state (or hazard). The hazardous software state (or hazard) is the top node of a fault tree, and the fault tree would be expanded from this node. However, the top node must be identified first by other techniques such as HAZOP, FMEA, etc.

Step 2. Identify the module of the SMV model related to the hazard. Select the module that generates outputs related to the hazard among all modules of the SMV model.

Step 3. Specify ‘output error’ of the module (identified in step 2) on the top node. ‘Output error’ means that incorrect and unexpected outputs are wrongly generated although the condition of generation of other outputs is satisfied.

Step 4. Specify ‘module error’ (identified in step 2) on the node directly connected to the top node. ‘Module error’ means that the corresponding module causes an ‘output error’.

Step 5. Extend the fault tree node with ‘input error’ (I.E.) and ‘transition error’ (T.E.). ‘Input error’ means that wrongly generated inputs cause a ‘module error’. An ‘input error’ of a module can be an ‘output error’ of the following modules. ‘Transition error’ means that a logical error (including a predefined value error) found inside a module causes an ‘output error’, and extension of this part heavily depends on the ability and intention of the analyst.

Step 6. Specify ‘module error’ again on the node connected to the directly higher level node.

3. Safety verification by model checker SMV

For automatic verification of safety properties, we use a model checker SMV, and properties to be checked are usually expressed in computation tree logic (CTL) [9,10]. CTL is a branching time temporal logic, extending propositional logic with temporal operators that express how propositions change their truth values over time. In this paper, we only use two temporal operators, A and G, or more precisely, a combinator AG, which expresses safety properties in the most natural way. Each CTL formula is evaluated with respect to some particular state. The formula AG p holds in state s if p holds in all states along all computation paths starting from s, and we call such a property an invariant. CTL formulae are implicitly evaluated by SMV with respect to all initial states.

Let $P_n$ be the CTL formula semantically equivalent to the event described in a fault tree node $n$. The property $AG (P_n)$, which means that for all states of system, the property $P_n$ is always untrue, i.e. node $n$ event never occurs, determines if the system can ever reach such a state. If the property is satisfied, the state denoted by $P_n$ will never occur, and the system is free from this

\[\begin{align*}
& C \\
& D \\
& E \\
& A \\
& B \\
& F
\end{align*}\]

Fig. 3. A simple system model.

Step 7. Continue on step 5 and 6 until no module generates outputs.

Step 8. Specify the error of the initial inputs on the leaf node, and connect to the directly lower level node. ‘Initial input error’ means a ‘type/range mismatch error’ of the initial input.

As an example, the system model shown in Fig. 3 is considered. The system model consists of two modules (A and B) in which state transition relations and transition constraints of the system are described by SMV input language according to the requirements of the system, three initial inputs (C, D and E), and a final output (F). Module A has two inputs; one is an output of module B and the other is an initial input (E), and generates the final output (F). Module B has two initial inputs (C and D), and generates an output that becomes an input of module A.

Before synthesis of a fault tree in accordance with the proposed procedure, the hazard is assumed to be the state that the final output (F) is generated incorrectly and unexpectedly. Module A is selected at step 2 because module A generates the final output (F) that is related to the hazard. The top node of the fault tree becomes ‘Output F error’ at step 3. At step 4, ‘Module A error’ node is added directly under the top node, and then the fault tree is extended by ‘input error’ (I.E.) and ‘transition error’ (T.E.). Modules A and B have child nodes named ‘logical error’ by ‘transition error’ according to step 5. Because the output of the module B becomes the input of module A, ‘Module B error’ node is connected to ‘Module A error’ node at step 6. ‘Input C error’ node and ‘Input D error’ node are added to ‘Module B error’ node, and ‘Input E error’ is added to ‘Module A error’ node at step 8. The corresponding fault tree generated by the proposed procedure is shown in Fig. 4.

\[\begin{align*}
& \text{A safety property expresses that, under certain conditions, an event never occurs. In general, safety statements express that an undesirable event will not occur, hence the “safety property” terminology [11]. For example, we can state “memory overflow will never occur” by AG overflow in CTL.}
\end{align*}\]
hazard. If, on the other hand, the property is not satisfied, the event denoted by $P_0$ is indeed possible. When the property is not satisfied, SMV generates a trace of a counterexample that provides detailed (but partial) information on how such a hazard may occur. Detailed analysis of the counterexample may provide useful information. Therefore, the corresponding suggestion for safety should be reflected to the system in order to eliminate this cause of the top event, which is the hazard of the system.

### 4. Case study: safety analysis on Wolsong SDS2

To demonstrate the feasibility of our approach, the proposed analysis method was applied to SDS2 of Wolsong NPP [13,14]. We selected Wolsong SDS2, which corresponds with the target system of [15], as our target system, because the fault tree in [15] has been verified, and hence is reliable and trustable. Therefore, we use this fault tree for comparison of a fault tree from natural language with a fault tree from the SMV model for a more reliable comparison.

SDS2 is the second of two independent emergency shutdown systems designed to rapidly terminate the nuclear chain reaction and then keep the reactor in a sub-critical state following detection of an unsafe operating condition. SDS2 operates by de-energizing relays, after a trip condition is detected by the trip logic that initiates the opening of quick opening valves (QOVs), which cause high-pressure helium to push neutron-absorbing liquid poison from tanks into the core via injection lines in the moderator. The SDS2 process parameter trip decisions have been implemented in software-driven programmable digital comparator (PDCs), and the neutronic-based trips (i.e., high neutron overpower and log rate) in hardware devices. The process trip parameters for SDS2 are as follows: pressurizer low level (PLL), steam generator low level (SGLL), primary heat transport low core differential pressure, primary heat transport low pressure (PHTLP), primary heat transport high pressure (PHTHP), and steam generator feedline low pressure (SGFLP). The trip decision for SDS2 is made using general coincidence two-out-of-three channel voting logic. In a given channel, the process trips have been implemented by dividing them between two PDCs labeled PDC1 and PDC2. Among the six trip parameters, we selected the primary heat transport low core differential pressure (PDL) trip condition as a target of the case study as it was used as an example in [15]. An overview of the PDL trip process is shown in Fig. 5 and high-level requirements for PDL trip were written in English in a document called the Program Functional Specification (PFS), as shown partially below:

#### 5.3.1. PHT Low Core Differential Pressure (68334)

The PHT Low Core Differential Pressure ($\Delta P$) trip parameter includes both an immediate and a delayed trip setpoint. Unlike other parameters, the $\Delta P$ parameter immediate trip low-power conditioning level can be selected by the operator. A handswitch is connected to a D/I, and the operator can choose between two predetermined low-power conditioning levels.

The PHT Low Core Differential Pressure trip requirements are:

- Determine the immediate trip conditioning status from the conditioning level D/I as follows:
  - If the D/I is open, select the 0.3% FP conditioning level. If $\phi_{LOC} < 0.3\%$ FP—50 mV, condition out the immediate trip.
  - If the D/I is closed, select the 0.3% FP conditioning level. If $\phi_{LOC} > 0.3\%$ FP, enable the trip. Annunciate the immediate trip conditioning status via the PHT $\Delta P$ trip inhibited ($\phi_{LOC} < 0.3\%$ FP) window D/O.
  - If the D/I is closed, select the 5% FP conditioning level. If $\phi_{LOC} < 5\%$ FP—50 mV, condition out the immediate trip.
  - If the D/I is closed, select the 5% FP conditioning level. If $\phi_{LOC} < 5\%$ FP, enable the trip. Annunciate the immediate trip conditioning status via the PHT $\Delta P$ trip inhibited ($\phi_{LOC} < 5\%$ FP) window D/O.
  - If the D/I is closed, select the 50% FP conditioning level. If $\phi_{LOC} < 50\%$ FP, enable the trip.
  - If the D/I is closed, select the 50% FP conditioning level. If $\phi_{LOC} < 50\%$ FP, enable the trip.

- If any $\Delta P$ signal is below the delayed trip setpoint and $\phi_{AVEC}$ exceeds 70% FP, open the appropriate loop trip error message.
If no PHT ΔP delayed trip is pending or active, then execute a delayed trip as follows:

i. Continue normal operation without opening the parameter trip D/O for normally three seconds. The exact delay must be in the range [2.7, 3.0] seconds.

ii. After the delay period has expired, open the parameter trip D/O if \( \phi_{AVEC} \) equals or exceeds 70% FP. Do not open the parameter trip D/O if \( \phi_{AVEC} \) is below 70% FP.

iii. Once the delayed parameter trip has occurred, keep the parameter trip D/O open for one second (±0.1 seconds), and then close the parameter trip D/O once all ΔP signals are above the delayed trip setpoint or \( \phi_{AVEC} \) is below 70% FP.

iv. Close the loop trip error message D/O if the signal is above the delayed trip setpoint or \( \phi_{AVEC} \) is below 70% FP.

Immediate trips and delayed trips (pending and active) can occur simultaneously. The loop and parameter trip D/Os are open if either an immediate or a delayed trip occurs.

In modeling the PDL trip process, some variables used in this work follow the convention of the four variable approach, and the prefixes \( m_\_ \), \( c_\_ \), and \( k_\_ \) represent monitored variables, controlled variables, and constant values, respectively. Other variables follow convention used in the NuSCR [16,17] approach, and the prefixes \( f_\_ \), \( h_\_ \), \( th_\_ \), and \( g_\_ \) represent function variable, history variable, timed history variable, and group variable, which is a set of function, history, and timed history variable, respectively. In addition, the prefix \( M_\_ \) is used for representing the module name in this work.

The PDLTrip process in Fig. A5 of Appendix A determines the value of \( f_{PDLTrip} \). The state of PDLTrip is determined by \( f_{PDLDly} \), \( f_{PDLSnrl} \), and \( f_{PDLCond} \). A PDL parameter trip results from an immediate trip or delayed trip. A combination of \( f_{PDLSnrl} \) and \( f_{PDLCond} \) determines an immediate trip and \( f_{PDLDly} \) determines a delayed trip. If \( f_{PDLSnrl} = k_{SnrTrip} \) and \( h_{PDLCond} = k_{Condln} \), the immediate trip condition becomes true, and if either an immediate trip or delayed trip occurs, PDLTrip condition is satisfied; i.e. \( f_{PDLTrip} \) is assigned to \( k_{Trip} \).

All modules to describe the PDL trip process are modeled in a similar manner by SMV input language in Appendix A.

4.1. Fault tree synthesis

In this section, we compare a fault tree constructed from natural language with a fault tree from a SMV model. Based on the results of the comparison, we demonstrate the feasibility and utility of the proposed approach, i.e., synthesis of a fault tree from a SMV model. In particular, we consider two points: (i) a fault tree from a SMV model can cover all contents or information of a fault tree from natural language; and (ii) the proposed approach is helpful for reducing errors that can occur in the process of synthesis of a fault tree from natural language. The fault tree in [15] is used as a fault tree for comparison with the fault tree from the SMV model, because this fault tree has been verified, and thus is reliable and trustable. The fault tree in [15] is shown in Fig. 6, and that from the SMV model in accordance with the proposed procedure is shown in Fig. 7. Expanded fault trees of transfer A and B are shown in Figs. 8 and 9, respectively.

To demonstrate that the fault tree from the SMV model can cover all contents or information of the fault tree from natural language, node numbers are tagged on each node of the fault tree from the SMV model. Nodes with same node numbers have the same meaning. For example, ‘node 1’ in Fig. 7, which reads ‘(\( f_{PDLTrip} = k_{NotTrip} \)) error’, corresponds with ‘node 1’ event in Fig. 6. The description of ‘node 1’ in Fig. 6, ‘PDL trip fails to clear trip D/O in required time’, denotes a situation where PDL trip does not occur (PDL trip fails to clear trip D/O) although the trip condition becomes true (required time). The ‘M_\_f_{PDLTrip}’.
Fig. 7. Fault tree synthesized from SMV model.

Fig. 8. Expansion of transfer A.
module determines PDL trip and generates an output, either ‘f_PDLTrip = k_Trip’ or ‘f_PDLTrip = k_NotTrip’, which means ‘PDL trip occurs’ and ‘PDL trip does not occur’, respectively. According to the definition of ‘output error’ in Section 2.3, ‘(f_PDLTrip = k_NotTrip) error’ has the meaning that PDL trip does not occur although the trip must occur, i.e. the trip condition is satisfied. Therefore, ‘node 1’ in Fig. 6 and ‘node 1’ in Fig. 7 have the same meaning.

In [15], it was found that colored nodes in Fig. 6, ‘node 3’ and ‘node 12’, have errors. According to [15], ‘node 3’ is incorrect because it does not consider all possible cases, and hence is incomplete. Therefore, ‘node 3’ must be changed to ‘Trip signal is turned off (e.g. becomes NotTrip) when the condition of one trip becomes false although the condition of the other continues to be true’ (or two separate nodes can be drawn). The corrected result for ‘node 3’ is shown in Fig. 10. The defective ‘node 3’ is modified to a new ‘node 3’ of the corrected fault tree, and ‘node 3.2’ is added for completeness.
However, because the SMV model in Fig. 11 describes all trip conditions clearly and definitely (e.g. immediate trip condition \( f_{PDLSnr} = k_{SnrTrip} \) & \( h_{PDLCond} = k_{CondIn} \)) and delayed trip condition \( (th_{PDLdly} = k_{InDlyTrip}) \), it is expected that the analyst can easily consider all cases without big efforts. Therefore, we can conclude that the formal nature of the SMV model can be helpful to reduce this kind of error.

‘Node 12’ in Fig. 6 is also defective, because the description of ‘node 12’ is not sufficient, i.e. it initially contains an implicit assumption. The specification in Fig. 12 denotes the assumption, ‘If any AP signal is below the delayed trip setpoint and \( \phi_{AVEC} \) exceeds 70% FP’, and therefore the description of ‘node 12’ itself is meaningless and insufficient. The corrected result for ‘node 12’ is shown in Fig. 13. In the same manner as the ‘node 3’ case, we can also conclude that the SMV model in Fig. 14 may reduce this kind of error with its formality.

4.2. Safety verification and analysis

4.2.1. First case: node 3.1

Model checking formula \( AG \sim (P_3) \) is used for checking the existence of the ‘node 3.1’ event, i.e. checking whether the ‘node 3.1’ event in Fig. 8 can occur. For this, the property to represent ‘node 3.1’ should be made in the form of CTL to use SMV. The property is verified by SMV and finally an analysis of the verification result is performed.

\![\neg (th_{PDLdly} = k_{InDlyTrip}) \] denotes that the delayed trip condition becomes false, and \( (f_{PDLSnr} = k_{SnrTrip}) & (h_{PDLCond} = k_{CondIn}) \) denotes the activation of the immediate trip condition, and \( (f_{PDLtrip} = k_{NotTrip}) \) denotes that PDL trip does not occur. Therefore, ‘node 3.1’ \( (P_{3.1}) \) means that the trip signal is turned off when the delayed trip conditions become false although the immediate conditions continue to be true, as described in ‘node 3.1’ in Fig. 10. The corresponding property translated into CTL expression is as follows:

\[
AG \sim (\neg (th_{PDLdly} = k_{InDlyTrip}) \land (f_{PDLtrip} = k_{NotTrip}))
\]

The verification result is “Property is not satisfied”. When properties are not satisfied, SMV provides a trace of a counter example. The screen of the trace is shown in Fig. 15. The trace reveals that the event of ‘node 3.1’ can occur, and details are as follows: Before step 10, a combination of the values of variables satisfies the conditions of an immediate trip, a delayed trip and a parameter trip. However, from step 10, the values are changed, and eventually the conditions for occurrence of the event of ‘node 3.1’ are satisfied. At step 10, the value of \( f_{FaveC} \) is changed from

---

**Fig. 12.** Natural language specification corresponding with the node 12.

**Fig. 13.** Corrected result for the node 12.

**Fig. 14.** SMV model corresponding with the node 12.

**Fig. 15.** Verification result of the node 3.1 (trace screen for the counter example).
step 13, which means the value of $h_{\text{PDLC}}$ changes to 0 at step 13. At step 14, the change of the value of $f_{\text{Flog}}$ from 327 to 0 at step 14, and the value of $h_{\text{PDLC}}$ changes from 1 at step 13 to 0 at step 14. At the final step, the values of four variables, $f_{\text{PDL}}$, $h_{\text{PDLC}}$, and $f_{\text{PDLC}}$, have important meaning. The values of $f_{\text{PDL}}$ and $h_{\text{PDLC}}$ become 0 simultaneously denoting the activation of the immediate trip condition, and the value of $h_{\text{PDLC}}$ becomes 1 denoting that the delayed trip condition becomes false. The value of $f_{\text{PDLC}}$ becomes 1 meaning that the condition of parameter trip is activated. Therefore, the boxed values of the four variables in Fig. 15 mean that the trip signal is turned off when the delayed trip conditions become false although the immediate conditions continue to be true, as described in 'node 3.1' in Fig. 10. By the

```
MODULE M_f_PDLSpI(f_PumpMde)
VAR
    f_PDLSpI : 0.2000;
ASSIGN
    next(f_PDLSpI) := case
        f_PumpMde = k_PumpAll : k_PDLSpIAll;
        f_PumpMde = k_Pump13 : k_PDLSpI13;
        f_PumpMde = k_Pump24 : k_PDLSpI24;
    1 : f_PDLSpI,
    esac;
DEFINE
    --Constants
    k_PDLSpIAll := 1710;
    k_PDLSpI13 := 1275;
    k_PDLSpI24 := 1350;
```

Fig. A1. SMV model for f_PDLSpI module.

```
MODULE M_f_PDLSpI(f_PumpMde)
VAR
    f_PDLSpI : 0.2000;
ASSIGN
    next(f_PDLSpI) := case
        f_PumpMde = k_PumpAll : k_PDLSpIAll;
        f_PumpMde = k_Pump13 : k_PDLSpI13;
        f_PumpMde = k_Pump24 : k_PDLSpI24;
    1 : f_PDLSpI,
    esac;
DEFINE
    --Constants
    k_PDLSpIAll := 1710;
    k_PDLSpI13 := 1275;
    k_PDLSpI24 := 1350;
```

Fig. A2. SMV model for f_PDLSpI module.

```
MODULE M_f_PDLSpI(f_PumpMde)
VAR
    f_PDLSpI : 0.2000;
ASSIGN
    next(f_PDLSpI) := case
        f_PumpMde = k_PumpAll : k_PDLSpIAll;
        f_PumpMde = k_Pump13 : k_PDLSpI13;
        f_PumpMde = k_Pump24 : k_PDLSpI24;
    1 : f_PDLSpI,
    esac;
DEFINE
    --Constants
    k_PDLSpIAll := 1710;
    k_PDLSpI13 := 1275;
    k_PDLSpI24 := 1350;
```

Fig. A3. SMV model for h_PDLCond module.

128 to 0. This induces changes of the values of $f_{\text{PDLSpI}}$, $h_{\text{PDLSpI}}$ and $h_{\text{PDLC}}$ from 0 to 1 and from Pending to DlyTrip, respectively at step 11. The change of the value of $f_{\text{PDLSpI}}$ from 0 to 1 at step 11 induces changes of the value of $h_{\text{PDLSpI}}$ from 0 to 1 and from Condout to DlyNorm, respectively at step 12. A change of the value of $f_{\text{Flog}}$ to 200 at step 12 results in a change of the value of $h_{\text{PDLCond}}$ to 1 at step 13.
analysis of the trace, we can conclude that the hazard of the top node event caused by ‘node 3.1’ event is indeed possible. Paragraph ‘h’ in PFS describes only the condition of parameter trip(PDL_trip) initiation according to the occurrences of immediate trip and delayed trip, not any condition of parameter trip-off. We can draw inference from the paragraph ‘h’ that if one of immediate trip and delayed trip occurs and then is eliminated, the parameter trip turns off. But we can not get any information of parameter trip when both of immediate trip and delayed trip occur and then one of both turns off and therefore ‘node 3.1’ event is possible. To take follow-up measure against ‘node 3.1’ event, the description of parameter trip-off condition should be added to paragraph ‘h’ in PFS.

4.2.2. Second case: node 11

‘Node 11’ describes a failure mode where the system incorrectly clears a delayed trip signal outside the specified time range of [2.7,3.0 s]. Because the SMV accepts only integers as the value of the clock variable, time_1 in this example, 27 and 30 are used to indicate the required time zone. ((f_PDLSnrDly = k_SnrTrip) & (time_1 <= 27 & time_1 <= 30) & (f_FaveCPDL)) denotes conditions for the delayed trip, and !(th_PDLDly = k_InDlyTrip) denotes that the delayed trip condition becomes false. Therefore, ‘node 11’ (P11) means that the delayed trip fails although the delay time is in the range [2.7,3.0 s],

as described in ‘node 11’ in Fig. 13. The corresponding property translated into CTL expression is as follows:

\[
\text{AG} (! ((f_PDLSnrI = k_SnrTrip) & (h_PDLCond = k_CondIn)) & (th_PDLDly = k_InDlyTrip)) \]

The verification result is ‘Property is satisfied’. When the property is satisfied, the SMV provides only the result that the property is true. The result of verification is shown in Fig. 16. We can conclude from the verification result that the ‘node 11’ event cannot occur, and therefore the system is free from the hazard of the top node caused by the ‘node 11’ event.

5. Summary and conclusion

In this work, we proposed a new approach to resolve problems of the conventional method for safety analysis of software
requirements. When synthesizing a fault tree from natural language, errors can arise because of the informality of natural language as well as that of the fault tree. However, this problem can be resolved by using a formal model, instead of natural language specification, i.e., the formal nature of the formal model is helpful to reduce errors of the fault tree. Since automatic verification from the formal model is possible, the problem of manual safety verification can also be resolved.

A SMV system was used for our approach because it supports both input language for system modeling and a model checker for verifying system properties. In other words, a fault tree is synthesized from a SMV model and safety properties are verified by a model checker SMV in our approach. A synthesis procedure to synthesize a fault tree from a SMV model was suggested, and CTL formulas were proposed for verification of safety properties.

To demonstrate the feasibility of our approach, the proposed analysis method was applied to shutdown system 2 (SDS2) of Wolsong nuclear power plant (NPP). In this case study, we compared a fault tree from natural language with a fault tree from a SMV model considering two points; whether the fault tree from the SMV model can cover all contents (or information) of the fault tree from natural language and whether our approach is helpful for reducing errors that can occur in the process of synthesis of the fault tree from natural language. We used the fault tree in[15] as a fault tree for comparison with the fault tree from the SMV model, because of its reliability and creditability. Through an example of the ‘node 1’ case, it was demonstrated that the fault tree from the SMV model can cover all contents (or information) of the fault tree from natural language. To verify that our approach is helpful for reducing errors in the fault tree, the cases of ‘node 3’ and ‘node 12’, which are defective, were analyzed. Based on the results, it was concluded that the formal nature of the SMV model can be helpful to reduce these kinds of errors.

Model checking formulas were used for checking the existence of ‘node 3.1’ and ‘node11’ i.e. checking whether events of these nodes can occur. The verification result for ‘node 3.1’ was ‘Property is not satisfied’. By the analysis of the trace, we can conclude that the hazard of the top node event caused by ‘node 3.1’ events is indeed possible, and therefore follow-up measures should be taken. The verification result for ‘node 11’ was ‘Property is satisfied’. Based on this result, we can conclude that the ‘node 11’ event cannot occur, and therefore the system is free from the hazard of the top node caused by this event. Based on the results

---

Fig. A8. SMV model for f_PDLCondLA module.

Fig. A9. SMV model for f_PDLCondHA module.

Fig. A10. SMV model for main module.
of this case study, the feasibility and utility of the proposed approach has been demonstrated.

Appendix A

This appendix describes all modules, which are modeled by SMV input language, for the PDL trip process (see Figs. A1–A10).

References