Future Generation Computer Systems 28 (2012) 1272–1282

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

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

A safety-focused verification using software fault trees[☆]

Sungdeok Cha[a], Junbeom Yoo[b,*]

[a] Korea University, Department of Computer Science and Engineering, Anam-dong Seongbuk-gu, Seoul 136-701, Republic of Korea

[b] Konkuk University, Division of Computer Science and Engineering, 1 Hwayang-dong, Gwangjin-gu, Seoul 143-701, Republic of Korea

A R T I C L E I N F O

Article history:
Received 30 October 2010
Received in revised form
7 January 2011
Accepted 3 February 2011
Available online 3 March 2011

Keywords:
Safety analysis
Software fault tree
Software verification
Combinational equivalence checking

A B S T R A C T

When developing safety-critical software such as reactor protection systems (RPS) in nuclear power plants, a demonstration of software trust (e.g., safety) is not only absolutely essential but also usually mandated by government authorities. While automated generation of fault trees has become possible with increased use of formal specifications, industrial use of fault trees has been limited primarily to safety demonstrations that the system is free from behavior captured in the root node. In this paper, we propose to extend the use of automated fault tree for verification purposes. As a fault tree represents an abstract and partial behavioral model of software on credible causes leading to a hazard, it must still satisfy various properties (e.g., fairness, correctness). Verification of a fault tree is useful when developing safety-critical software because (1) it strengthens a safety-focused software development process; (2) it provides an opportunity to detect potentially critical errors early; and (3) it is less likely to experience a state explosion problem. This paper demonstrates how to convert a fault tree into a semantically equivalent logic formula so that they can be subject to formal verification using tools like Verification Interacting with Synthesis (VIS). We evaluated the feasibility of FTA's applicability as a verification tool on a prototype model of a nuclear power reactor protection system (RPS) software to be deployed in plants under construction in Korea.

1. Introduction

Fault tree analysis (FTA) [1] is the most widely used safety analysis [2] technique when developing safety-critical real-time systems such as RPS (Reactor Protection System) in nuclear power plants. An independent safety demonstration is often mandated by regulatory agencies. FTA is essentially a logical argument, in graphical notation, that the system is free from the root-node failure. It is essentially a partial and abstract model of software behavior with respect to failure modes. Unfortunately, existing techniques still have intrinsic limitations in that fault trees contain only information explicitly captured in the specification or source code and that it rarely serves beyond safety demonstration purposes.

Increased use of formal specification has made mechanical (e.g., automated [3–6] or template-based [7,8]) construction of fault trees possible, and such advances create possibilities of extending the use of fault trees beyond safety demonstrations. In this paper, we propose to extend the use of mechanically constructed fault trees for verification purposes by applying a semantic preserving translation into a propositional logic formula. Contributions include refinement of software fault tree semantics and development of an algorithm to translate a fault tree into Verilog. Because a fault tree must still satisfy various properties (e.g., fairness, correctness, etc.), we use VIS's combinational equivalence checking [9] to automate verification. Should verification fail, we have detected a logical error in the fault tree. A counterexample provides tips on why properties were not met and how the behavioral model must be fixed.

We argue that fault tree-based verification of safety-critical behavior has the following advantages: (1) it encourages safety-focused development of software throughout life-cycle phases; (2) it provides opportunities to detect potential errors early and identify missing safety requirements; and (3) the state explosion problem is less likely to occur as verification is carried out using a partial (but relevant) behavioral model. In order to demonstrate effectiveness of our idea, we performed a case study using a prototype specification of KNICS[1] RPS BP (Bistable Processor) logic. The same system, to be installed in the nuclear power plants

[☆] This research was partially supported by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development and also by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency (NIPA-2010-(C1090-0903-0004) and NIPA-2010-(C1090-1031-0003)). This research was also supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0002566).

[*] Corresponding author. Tel.: +82 2 450 3258.
E-mail addresses: scha@korea.ac.kr (S. Cha), jbyoo@konkuk.ac.kr (J. Yoo).

[1] Korea Nuclear Instrumentation and Control System.

under construction in Korea, was used in other papers [10,11,5] too. Results indicate that fault tree-based verification can detect errors quicker than full-scale formal verifications do and that mechanically generated fault trees serve a useful role in safety verification too. Quantitative results obtained from preliminary FTA-based verification appear quite promising, and a full-scale evaluation using the entire KNICS RPS system is planned. It encompasses refinement of the semantics of software fault trees and mechanical translation into Verilog programs.

The rest of our paper is organized as follows. Section 2 reviews recent advances on software fault tree analysis techniques. Formal verification techniques with emphasis on VIS are also briefly reviewed. In Section 3, to help readers better understand the example, we explain the KNICS nuclear reactor protection system mechanism and describe fault trees constructed for the KNICS RPS system. It must be noted that the KNICS system used in this paper has passed operational "fitness" evaluation from the Korean regulation agency called KINS (Korean Institute of Nuclear Safety) and that the system is to be deployed in 2013. Research on this paper began after completing the KNICS RPS certification process. We also explain key features of NuFTA, a CASE tool being developed by our research group, with the focus on automated fault tree construction and Verilog translation. Section 4 shows how the safety-focused software verification process is applied. Section 5 describes several practical issues that must be addressed to apply the proposed idea to the real-world applications. Finally, Section 6 concludes the paper.

2. Background

2.1. Fault tree analysis techniques

To the best of our knowledge, recent advances on fault tree analysis techniques belong to one of the following categories: dynamic (temporal) analysis, formalization of FTA, compositional analysis, or mechanical fault tree construction.

Dynamic fault tree (DFT) [12] and its Galileo FTA tool [13] use Markov chains to analyze behavior of dynamic system events such as sequential dependencies, functional dependencies, or redundancy. Temporal extensions to fault trees [14], based on a temporal logic called Propositional Linear Temporal Logic (PLTLP), include a set of new gates such as UNTIL-PAST, PREV, and WITHIN. Similarly, temporal fault tree (TFT) methodology enables temporal analysis using gates such as TAND gate (i.e., AND.THEN). Other extensions include Priority-AND (PAND) gate [15].

Formalization of fault tree semantics is the necessary step for the mechanical generation of fault trees as well as their formal verification. [16] proposed a formal semantics of fault trees in temporal logic and defined the consistency relation between the system model and fault trees. [17] addressed how software safety requirements may be derived from the fault trees. It uses a system model common to both fault tree analysis and program development, but issues related to the decomposition problem of fault trees remain unaddressed. [18] proposed to use Interval Temporal Logic with continuous semantics as a means of defining fault tree semantics so that the decomposition problem can be better addressed. Similarly, [19] defined formal semantics of dynamic fault trees using Z specification language, but one can neither validate (e.g., via simulation) nor verify the correctness of safety requirements discovered by the fault trees. An executable algebraic specification (e.g., CafeOBJ [20]) can be used as an alternative to Z. Lastly, [21] reported practical experiences on application of fault tree analysis to infinite state systems based on fault tree semantics defined in [22].

Approaches to compositional FTA, essential in applying safety analysis on large and complex systems, include Failure Propagation and Transformation Notation (FPTN) [23] and HiP-HOPS [4]. The former illustrates how component modules describing the local generation and propagation of failures may be composed into larger system modules. The latter extends key concepts introduced in FPTN to include algorithms for synthesis of fault trees and Failure Mode and Effect Analysis (FMEA). A key difference between FPTN and HiP-HOPS is that failures in FPTN may propagate between FPTN modules while failures in HiP-HOPS propagate only via input and output parameters on connections between components. Component Fault Trees (CFTs) [24,25] and State-Event Fault Trees (SEFTs) [26,27] are successors to FPTN and HiP-HOPS techniques. In CFTs, local failure logic of components is defined in a graph of interconnected and modularized fault trees. SEFT enables representation of states in the failure logic, thereby allowing analysis of systems that exhibit dynamic failure behavior.

Mechanical construction of fault trees is the topic most relevant to the ideas described in this paper. Fault trees can be generated from source codes written in various languages (e.g., C, Ada83, Ada95, or Function Block Diagrams), requirements specification, or design documents. [7,28,29] defined fault tree templates for Ada83 or Ada95 constructs (e.g., if-then-else, procedure call, rendezvous, etc.) so that safety analysis can be guided by the relevant failure semantics of language constructs. [8,30] defined fault tree templates for Function Block Diagrams (FBDs) [31] which is one of the most widely used PLC (Programmable Logic Controller) programming languages. For example, FBD was successfully used in the development of the KNICS RPS system, and [5] proposed how fault trees can be synthesized from a formal requirements specification written in a language named NuSCR [11]. Likewise, [6,32,33] describes how fault trees can be mechanically generated from other formal specifications written in languages such as Statecharts [34] or RSML (Requirements State Machine Language) [35].

Fault tree construction on a design document is addressed in [4] in which fault trees are generated from an architectural diagram similar to a data flow diagram. It treats the internal logic behind each component in a black-box manner and analyzes interactions among software components. [36] extended the idea of fault tree generation to include Matlab–Simulink models. [3] is another approach for generating fault trees from an architecture description language called RIDL (Reliability Imbedded Design Language). An analysis tool named Galileo [37] was implemented, too. RIDL may express the impact of architectural decisions such as redundant modules and components on system reliability.

2.2. Formal methods and VIS

Formal method techniques encompass formal specification languages and formal verification techniques. The former addresses how mathematics or logic could be effectively used to describe software requirements or design without ambiguity, incompleteness or inconsistency. Various flavors (e.g., algebraic, tabular, graphical, logic-based, or automata-based) of formal specification languages have been proposed. Formal specification languages known to have been successfully used in industrial projects include Statechart [34], RSML [35], timed automata - UPPAAL [38], SCR [39], Petri-Nets [40], NuSCR [11], Z [41], process algebra [42] and SDL [43].

Approaches to formal verification generally belong to either deductive reasoning or algorithmic verification. Deductive reasoning [44] is a verification methodology in which human experts use axioms and proof rules to guide the reasoning process. Although modern state-of-the-art theorem provers such as PVS [45] provide features to automate certain degrees of reasoning, the success of the proof mostly depends on the technical expertise of analysts making critical decisions in building proof strategies.

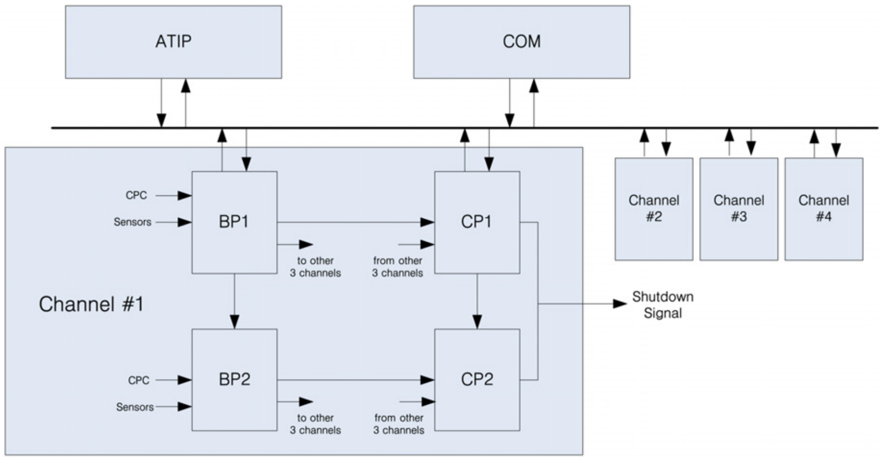


Fig. 1. Simplified architecture of the KNICS RPS.

Model checking [46–49] is widely used in industry because it exhaustively searches finite system space to examine if properties are always satisfied. Although the state explosion problem is still a practical obstacle in verifying the behavior of large and complex systems, rapid advances on model checking theories and tools have made it possible to verify many industrial systems. Representative model checking tools include SMV [48], Spin [50], VIS [9], and UPPAAL [38]. Code verification is also an active research topic, and tools such as CBMC [51] or BLAST [52] work directly on C programs.

VIS is the system we use in extending the use of fault trees as a verification aid. It integrates formal verifications, simulation, and synthesis of finite states hardware systems. It uses Verilog as a front end and supports fair CTL (Computational Tree Logic) model checking, language emptiness checking, combinational equivalence checking, sequential equivalence checking, cycle-base simulation and hierarchical synthesis. In the case of sequential equivalence checking, the Verilog program has *reg* type variables to store information internally. In our earlier work [53], we developed an algorithm to translate FBD into a semantically equivalent Verilog program and proved behavioral equivalence between the successive revisions. When developing safety critical software, such verification is important because minor modifications or code optimization may result in unexpected failures. We also developed VIS Analyzer [54] to provide a graphical interface so that domain experts may take full advantage of VIS's powerful features without being overwhelmed by a primitive and text-based interface. Nuclear engineers found the VIS Analyzer interface easy enough to use without having to rely on technical support from formal methods experts.

3. Software fault trees for the KNICS RPS

KNICS RPS is a digital system whose responsibility is to shutdown a nuclear reactor safely in case of emergency. It has been approved for operational fitness evaluation tests in two nuclear power plants being built in Korea. As a safety-critical system, it has four redundant and physically isolated channels to provide defense in depth. The high-level architecture diagram[2] for single channel RPS is shown in (Fig. 1). It consists of two bistable processors (BPs), two coincidence processors (CPs), an automatic test and interface processor (ATIP) and a cabinet operator module (COM). Subsystems are interconnected with different networks.

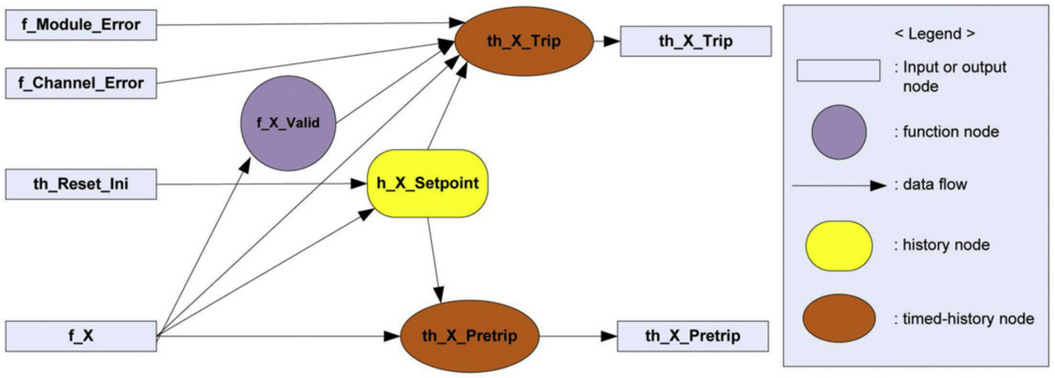
BP generates a trip signal to the CPs by comparing values of 18 process variables against predefined threshold values. There are four different trip logics built in the system: (1) fixed set-point trip (10 variables); (2) variable set-point trip (3 variables); (3) manual reset trip (3 variables); and (4) digital trip (2 variables). Upon receiving trip signals, CPs execute two-out-of-four voting logic to determine if the trip signal must be sent to the hardware actuators. All RPS channels are duplicated, and each one has two independent BPs and CPs respectively. ATIP, primarily used to execute for either manual or automated tests initiated by operators, interact with BP or CP in a single channel or multiple channels as a whole through the common bus. COM, located at an operator room and connected to other processors through the common bus, has two parts: (1) a computer-based unit which provides status information regarding the overall RPS equipment, and (2) a hardware unit which performs protection-related controls such as channel bypass and initiation circuit reset.

In the KNICS RPS, BP trip logics and CP voting logics replace traditional relay-based analog systems. They are classified as safety-critical units whose safety assurance demonstration is mandated. In order to fulfill regulatory requirements and enhance software quality, safety analysis was performed thoroughly [55] in addition to verification and validation (V & V) activities [56]. In particular, a formal specification language named NuSCR was used to describe RPS requirements.

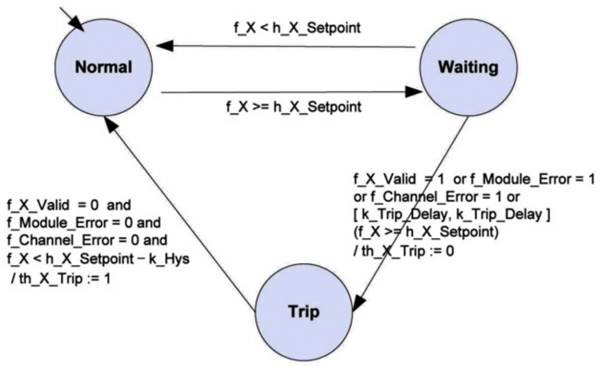
The NuSCR language has 4 basic constructs: FOD (Function Overview Diagram), FSM (Finite State Machine), TTS (Timed Transition System) [57] and SDT (Structural Decision Table). The FOD, shown in (Fig. 2(a)) allows hierarchical modeling of arbitrary depth. The proper prefix (e.g., '*g*' for a group node and '*f*' for a function node) is used to improve understandability of the specification. We use a partial specification of "manual reset variable set-point rising trip" logic in BP, shown in (Fig. 2), to illustrate our approach. (Fig. 2(b)) illustrates the NuSCR feature to specify real-time requirements using duration constraints embedded in transition conditions. We chose these notations in close consultation with domain experts while carefully balancing tradeoffs between the language's flexibility in expressiveness and capability to perform automated analysis.

If the system is in the '*Normal*', the default state, an immediate transition to the '*Waiting*' occurs when the trigger condition is satisfied. When the condition is augmented with timing constraints, as is the case in transition from the '*Waiting*' to '*Trip*' state, the condition must remain true for the specified duration. FSM is the same as TTS, but has no timing constraint. SDT,

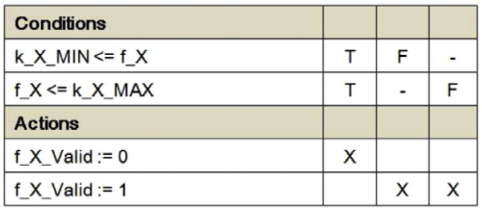
[2] (Fig. 1) represents a preliminary design. The certified design has been slightly modified.



(a) FOD for *g_Manul_Reset_Variable_Set_Point_Rising_Trip*.



(b) TTS for *th_X_Trip*.



Conditions			
k_X_MIN <= f_X	T	F	-
f_X <= k_X_MAX	T	-	F
Actions			
f_X_Valid := 0	X		
f_X_Valid := 1		X	X

(c) SDT for *f_X_Valid*.

Fig. 2. *g_Manul_Reset_Variable_Set_Point_Rising_Trip* logic in simplified NuSCR specification.

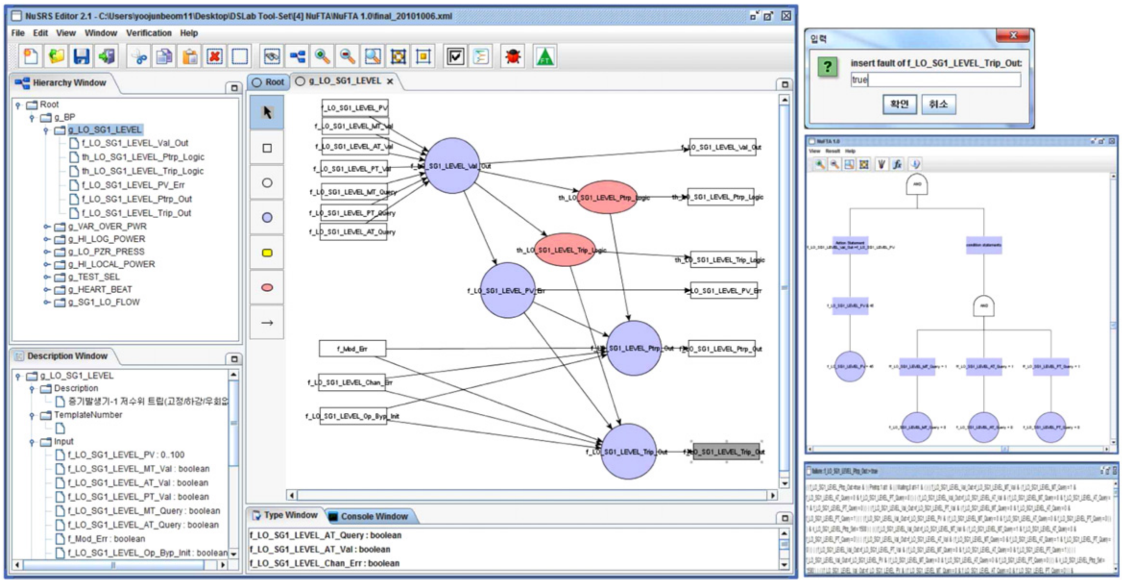

NuSRS Editor 2.1 - C:\Users\yoojunbeom11\Desktop\DSLab Tool-Set\[4] NuFTA\NuFTA 1.0\final_20101006.xml
File Edit View Window Verification Help
Hierarchy Window
Root
g_BP
g_LO_SG1_LEVEL
f_LO_SG1_LEVEL_Val_Out
th_LO_SG1_LEVEL_Ptrp_Logic
th_LO_SG1_LEVEL_Trip_Logic
f_LO_SG1_LEVEL_PV_Err
f_LO_SG1_LEVEL_Ptrp_Out
f_LO_SG1_LEVEL_Trip_Out
g_VAR_OVER_PWR
g_HI_LOG_POWER
g_LO_PZR_PRESS
g_HI_LOCAL_POWER
g_TEST_SEL
g_HEART_BEAT
g_SG1_LO_FLOW
Description Window
g_LO_SG1_LEVEL
Description
증기발생기-1 저수위 트립(고정/하강/우회
TemplateNumber
Input
f_LO_SG1_LEVEL_PV : 0..100
f_LO_SG1_LEVEL_MT_Val : boolean
f_LO_SG1_LEVEL_AT_Val : boolean
f_LO_SG1_LEVEL_PT_Val : boolean
f_LO_SG1_LEVEL_MT_Query : boolean
f_LO_SG1_LEVEL_AT_Query : boolean
f_Mod_Err : boolean
f_LO_SG1_LEVEL_Op_Byp_Init : boolean
Root
g_LO_SG1_LEVEL
Type Window
Console Window
f_LO_SG1_LEVEL_AT_Query : boolean
f_LO_SG1_LEVEL_AT_Val : boolean
f_LO_SG1_LEVEL_Chan_Err : boolean
입력
insert fault of f_LO_SG1_LEVEL_Trip_Out:
true
확인
취소
NuFTA 1.0

Fig. 3. A screen-dump of NuSCR specification of KNICS RPS BP in NuSRS and a fault tree produced by NuFTA.

organized in a style similar to the AND-OR table, specifies details on what action (e.g., assignment) must take place when. For example, the first column indicates that the *f_X_Valid* value becomes 0 when the *f_X* value lies in between (including the boundary) the two threshold values. Otherwise, it receives 1 as its value. For detailed description of NuSCR features and formal semantics, please consult [11]. (Fig. 3) shows the NuSCR specification of the KNICS RPS BP. The specification was developed using a NuSCR supporting tool NuSRS.

(Fig. 4) is a fault tree mechanically generated from the above NuSCR specification according to the synthesis procedure described in [5,58]. It illustrates credible causes and dependencies among them as to how the root node event, the *th_X_Trip* value becoming zero and thereby shutting down the reactor, may occur.

Table 1
Experiments on computation ability of NuFTA (range of variable x: 0–100, unit = s).

Monitor variables	Types of trip logic	Time to trip	Time to pre-trip
LO_SG1_LEVEL	Fixed	0.138	0.109
VAR_OVER_PWR	Variable	> 10 min	> 10 min
HI_LOG_POWER	Fixed	0.092	0.142
LO_PZR_PRESS	Manual reset	0.205	0.197
SG1_LO_FLOW	Manual reset	0.111	0.108
HI_LOCAL_POWER	Digital	0.008	0.004

Table 2
Experiments on scalability of NuFTA (for variable *LO_SG1_LEVEL*, unit = s).

Range of a variable	Time to trip	Time to pre-trip
0–10	0.096	0.093
0–100	0.138	0.109
0–1,000	0.351	0.170
0–10,000	3.377	0.453

Two possible causes are (1) unsafe condition may have been newly detected as indicated in the '*Waiting*' to '*Trip*' transition, and the detailed causes are shown in the condition nodes 1 and 2; or (2) the system could have already been in the '*Trip*' state, and the trigger condition to '*Normal*' state remains unsatisfied (e.g., condition 3).

NuFTA [58] is an assistant tool we continue to enhance as a part of the NuSRS tool-set for a software development framework based on formal methods [53]. It automates mechanical generation of fault trees and generates a propositional logic formula representing the minimal cut-set of the fault tree. NuFTA performs backward analysis starting from the system configuration corresponding to the root node event and continues until it finds all the credible combinations of input variables. The process is computation-intensive, and in general, the process can't be fully automated in theory. This is especially true when performing backward analysis on time-dependent failure modes.

As a practical alternative and a tool specialized for the nuclear engineering industry, NuFTA constructs a fault tree mechanically for the important safety-critical aspects of RPS design such as shutdown logic (e.g. trip or pre-trip signals). Our experience of using NuFTA on the preliminary KNICS design indicates that NuFTA can mechanically generate fault trees relatively quickly and that the fault tree was of an acceptable quality to safety engineers. We performed experiments using 6 of 18 monitor process variables, but the remains are their variants. Experimental results, shown in Tables 1 and 2, show that NuFTA completed fault tree generation on handling 15 monitor process variables in 18, or 3 out of 4 trip logics, almost instantly. An exception occurred when constructing a fault tree on *Var_Over_Pwr* logic, and the state explosion problem encountered during the execution of backward analysis is the most likely cause. We plan to enhance the fault tree generation algorithm. It must be stressed that the current NuFTA synthesis algorithm is focused more on correctness of its output and not on optimization of the fault trees.

4. Safety-focused verification using software fault trees

Given a software fault tree which can be mechanically generated from a formal specification, we now explain how one can formally verify it. As introduced earlier, we first translate the fault tree into a propositional logic formula using NuFTA and then into a Verilog program which is an input front-end of the VIS verification system. Properties that the requirements must satisfy are also abstracted from the fault tree analysis and translated into a Verilog program, too. Once two Verilog programs become available, VIS's combinational equivalence checking determines whether the model of the fault tree satisfies the model of requirements properties. (Fig. 5) describes an overview of the safety-focused verification using software fault trees.

4.1. Fault tree to Verilog translation

When translating a fault tree into a Verilog `module`, variables used in the root node are declared as `output` variables. Likewise, all the variables appearing in the leaf nodes (events) are declared as `input` variables. In addition, the Verilog module needs additional input variables to keep track of state configurations. The fault tree shown in (Fig. 4), constructed through two system execution cycles, needs two additional variables, '*Trip(t)*' and '*Waiting(t-1)*', to capture the current and previous system states, respectively. Assignments made to the output variable also depend on the pattern used in the AND/OR logic of the fault tree.

th_X_Trip

```
      = ¬ (th_X_Trip == 0)        // negation of the fault tree in ⟨ Fig. 4⟩
      = ¬ ( Trip(t) ∧ ( "Condition 1" ∨ "Condition 2" ∨ "Condition 3" ) )
      = ¬ ( Trip(t) ∧ "Condition 1" ) ∧ ¬ ( Trip(t) ∧ "Condition 2" ) ∧ ¬ ( Trip(t) ∧
"Condition 3" )
      =      // for Condition 1
        ¬ ( Trip(t) ∧ ( Waiting(t − 1) ∧ (f_X_Valid ∨ f_Module_Error ∨ f_Channel_Error)
) )
             // for Condition 2
        ∧ ¬ ( Trip(t) ∧ ( Waiting(t − 1) ∧ (f_X ≥ h_X_Setpoint) ) )
             // for Condition 3
        ∧ ¬ ( Trip(t) ∧ ( Trip(t − 1) ∧ ¬ ( ¬ f_X_Valid ∧ ¬ f_Module_Error ∧ ¬
f_Channel_Error
           ∧ (f_X < h_X_Setpoint - k_Hys) ) ) );
```

The Verilog module `SFT_Formula`, shown in (Fig. 6) is manually translated from the fault tree in (Fig. 4). It has 5 input variables, 1 output variable, and 2 state variables (i.e., '*Current_State*' and '*Previous_State*'). For convenience sake, we defined that two input variables may be assigned values ranging from 0 to 6 and that three input variables are of the Boolean type. The output variable *th_X_Trip* is assigned a new value as indicated in the logic formula of the fault tree above. In order to assist readers to better understand the mapping between the fault tree and Verilog program, illustrative comments are included. Unfortunately, propositional logic used in fault tree construction is unable to support all the features (e.g., time constraints expressed in durations) supported by NuSCR specification. As discussed in Section 5 below, we plan to revisit the technical challenge of extending the semantic definition of fault trees so that NuFTA and NuSCR specification languages may become fully compatible. In order to achieve this goal, other logic (e.g., duration calculus) may need to be used instead.

When performing fault tree analysis on the combination function, use of *reg* type variable is unnecessary because information on internal states need not be maintained. When safety analysis involves state-dependent information, one must apply sequential equivalence checking instead of combinational equivalence checking. In such a case, VIS model checking would take longer, and state explosion is more likely to occur.

4.2. Safety property to Verilog translation

A fault tree, as a partial model of system behavior with respect to the hazard, must still satisfy 'fairness' and 'correctness' properties. In the KNICS project, for example, the fairness property requires that a shutdown signal must be immediately generated if any input variable lies outside of the threshold values (e.g., predefined setpoints). The correctness property is useful in detecting potential logical errors in the fault tree. A correctness requirement may specify that "the shutdown signal should be produced (*th_X_Trip* := 0) only when the system is in the '*Trip*' state". For the fault tree shown in (Fig. 4), these properties must be shown to hold, and the corresponding logic formulae are shown below:

Fairness:

1. (F1) If the value of input variables is out of bounds, the system should fire a shutdown signal immediately (*th_X_Trip* := 0).

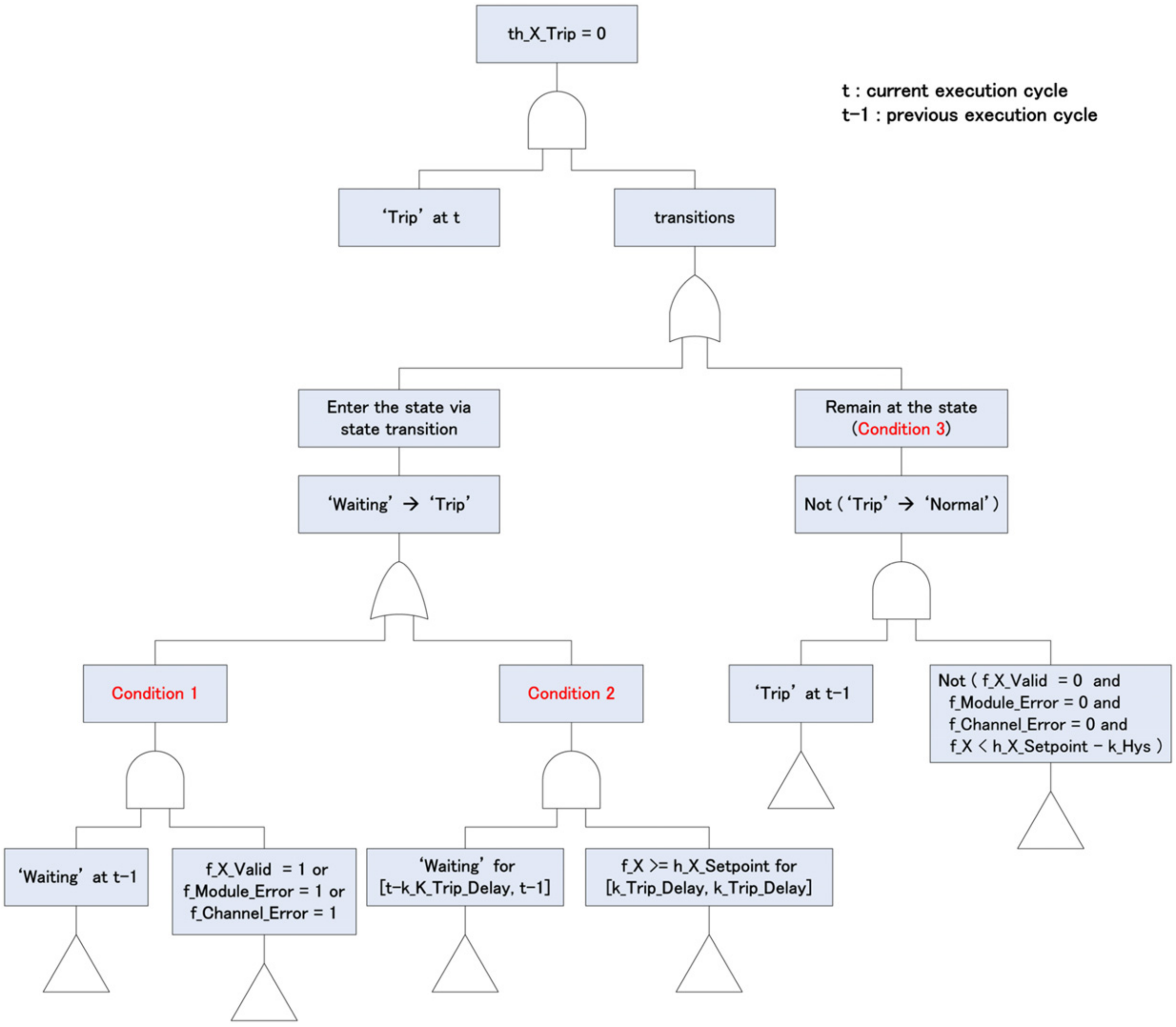

th_X_Trip = 0
t : current execution cycle
t−1 : previous execution cycle
'Trip' at t
transitions
Enter the state via state transition
Remain at the state (Condition 3)
'Waiting' → 'Trip'
Not ('Trip' → 'Normal')
Condition 1
Condition 2
'Trip' at t−1
Not (f_X_Valid = 0 and f_Module_Error = 0 and f_Channel_Error = 0 and f_X < h_X_Setpoint − k_Hys)
'Waiting' at t−1
f_X_Valid = 1 or f_Module_Error = 1 or f_Channel_Error = 1
'Waiting' for [t−k_K_Trip_Delay, t−1]
f_X >= h_X_Setpoint for [k_Trip_Delay, k_Trip_Delay]

Fig. 4. A software fault tree generated from the NuSCR specification in Fig. 2 (excerpted).

2. (F2) Only after all conditions resulting in the 'out of bounds error' cease to be true, it can cancel the shutdown signal.
3. (F3) If the trip condition regarding the operation variable *f_X* is satisfied, it should fire a shutdown signal immediately.
4. (F4) If the trip condition regarding the operation variable *f_X* is canceled, it should stop firing the shutdown signal immediately.

Correctness:

1. (C1) When the system enters the state '*Normal*', no shutdown signal should be fired.
2. (C2) When the system enters the state '*Waiting*', no shutdown signal should be fired.
3. (C3) When the system enters the state or remains at the state '*Trip*', a shutdown signal should be fired.

th_X_Trip =

$$
\begin{aligned}
&\neg(\neg Trip(t) \wedge (f\_X\_Valid \vee f\_Module\_Error \vee f\_Channel\_Error)) && //\,(F1)\\
&\vee(Trip(t) \wedge (\neg f\_X\_Valid \wedge \neg f\_Module\_Error \wedge \neg f\_Channel\_Error)) && //\,(F2)\\
&\vee \neg(\neg Trip(t) \wedge (f\_X \geq h\_X\_Setpoint)) && //\,(F3)\\
&\vee(Trip(t) \wedge (f\_X < h\_X\_Setpoint - k\_Hys)) && //\,(F4)\\
&\vee(Normal(t) \wedge \neg Normal(t-1)) && //\,(C1)\\
&\vee(Waiting(t) \wedge \neg Waiting(t-1)) && //\,(C2)\\
&\vee \neg(Trip(t) \wedge Trip(t-1) \vee \neg(Trip(t) \wedge \neg Trip(t-1)); && //\,(C3)
\end{aligned}
$$

Encoding the safety properties in Verilog, shown in (Fig. 7), is almost identical to the formula. Output variable *th_X_Trip* is assigned with the value 1 or 0, but in cases that no safety properties are concerned, it generates a safe (i.e., 1) in the same ways as the SFT_Formula. As the safety property specification did not use *reg* variable, VIS's combinational equivalence checking may be applied.

In the case study, VIS combinational equivalence checking, executed in the Cygwin environment, found that "*Networks are combinationally equivalent*". One must enter a series of commands (e.g., '*flatten_hierarchy*' and '*build_partition_mdds*'). While line-by-line explanation of the results shown in (Fig. 8) is unnecessary and is clearly outside the scope of the paper, we note the following:

- The counterexample described a scenario which led to different system behavior. Details were sometimes missing (e.g., variables whose values remain unchanged in successive states are not shown) and information display is primitive (e.g., *f_X* is an integer value but shown in bit vector format), for nuclear safety engineers, it was nearly impossible to accurately understand the root cause of different behaviors although VIS is a very powerful verification.
- Current VIS implementation provides only a command-based interface. Most domain experts (e.g., nuclear engineers) have neither expertise on formal methods to take full advantage nor the patience (or necessary resources) in reassembling missing information to pinpoint the cause of behavioral inequivalence.

(Fig. 9), *VIS_Analyzer* (version 3.0), is a tool we developed to provide a more user-friendly interface. Using the GUI, one can choose input files, launch a behavioral equivalence checking task by executing a series of commands in the background, and display

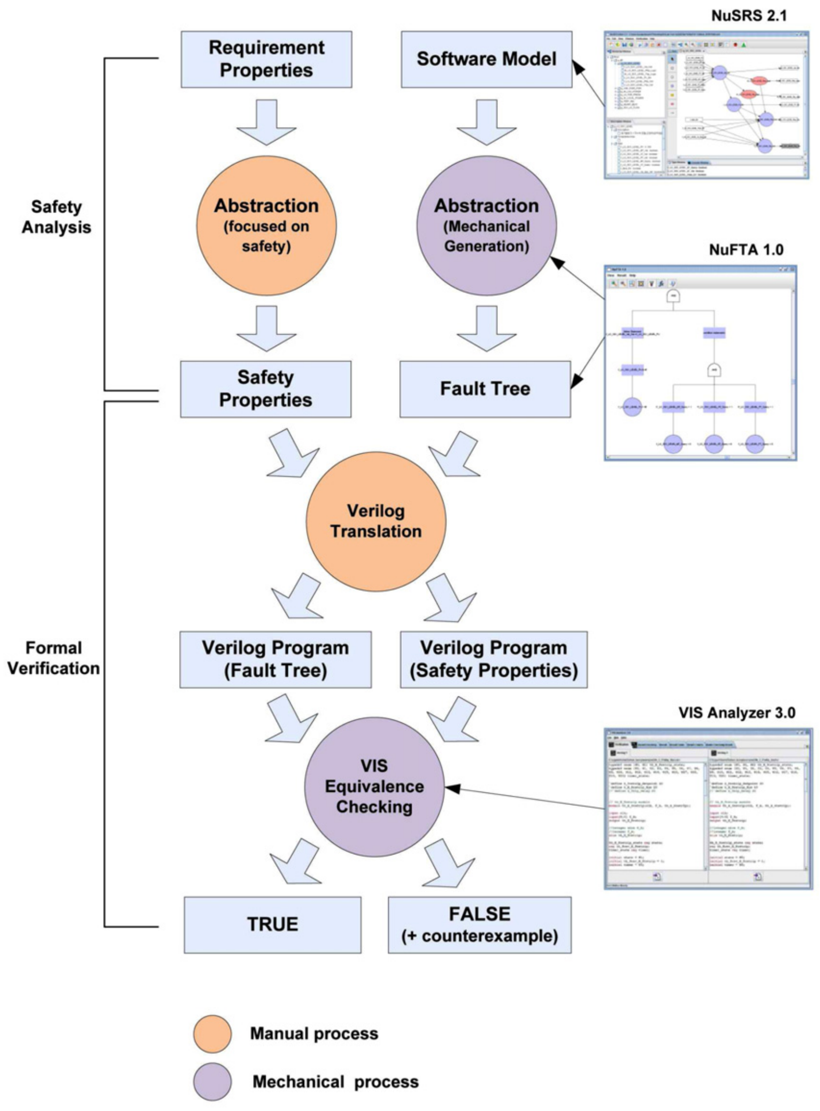
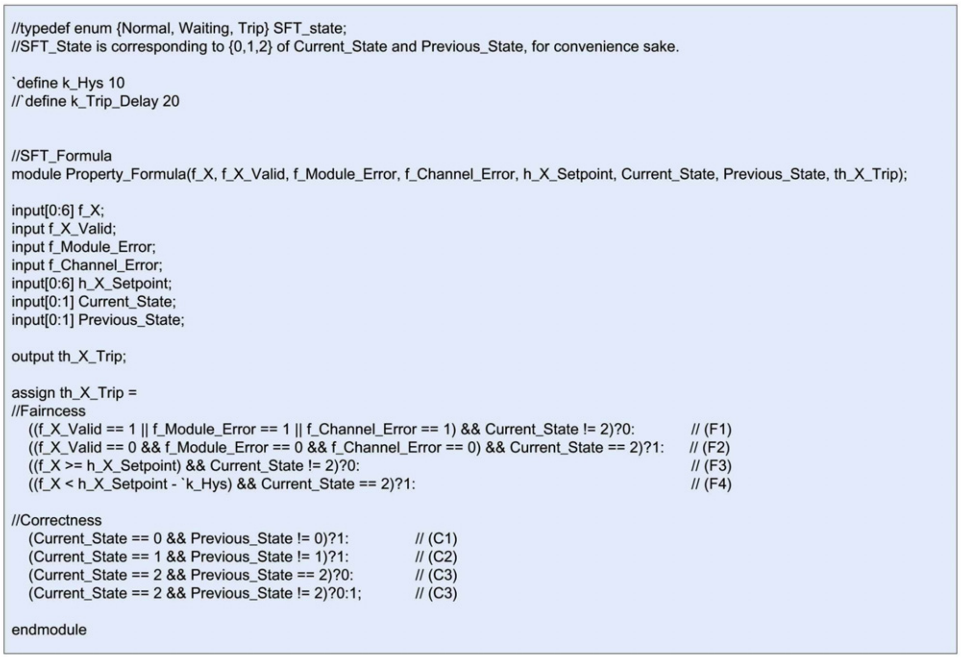


Fig. 5. An overview of safety-focused verification using software fault trees.

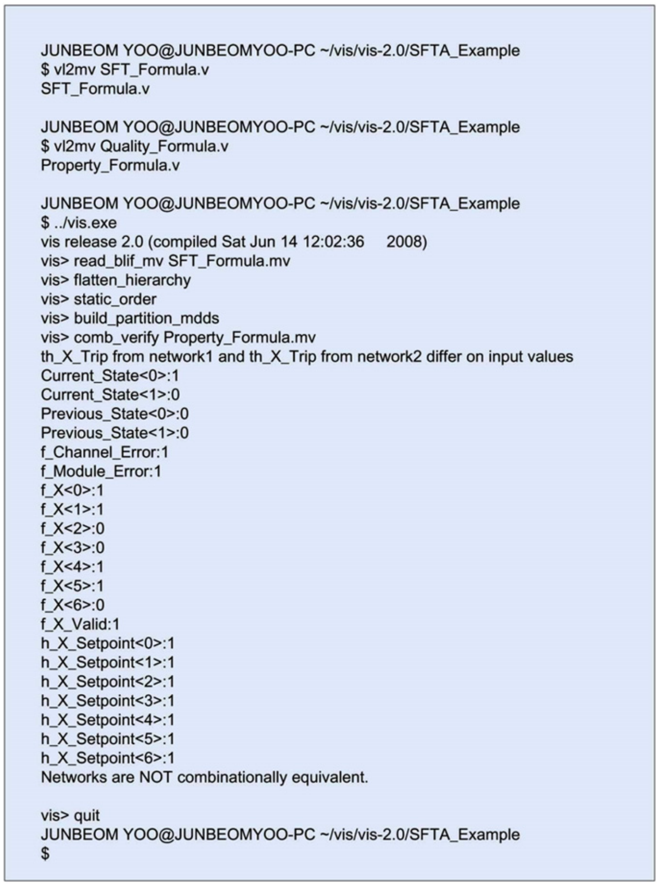
```
//typedef enum {Normal, Waiting, Trip} SFT_state;
//SFT_State is corresponding to {0,1,2} of Current_State and Previous_State, for convenience sake.

`define k_Hys 10
//`define k_Trip_Delay 20


//SFT_Formula
module SFT_Formula(f_X, f_X_Valid, f_Module_Error, f_Channel_Error, h_X_Setpoint, Current_State, Previous_State, th_X_Trip);

input[0:6] f_X;
input f_X_Valid;
input f_Module_Error;
input f_Channel_Error;
input[0:6] h_X_Setpoint;
input[0:1] Current_State;
input[0:1] Previous_State;

output th_X_Trip;


assign th_X_Trip =
   (Current_State == 2 && Previous_State == 0 && (f_X_Valid == 1 || f_Module_Error == 1 || f_Channel_Error == 1))?0:    // Condition 1
   (Current_State == 2 && Previous_State == 1 && (f_X >= h_X_Setpoint))?0:                                              // Condition 2
   (Current_State == 2 && Previous_State == 2 && !(f_X_Valid == 0 && f_Module_Error == 0 && f_Channel_Error == 0         // Condition 3
          && (f_X < h_X_Setpoint - `k_Hys)))?0:1;


endmodule
```

Fig. 6. A Verilog program translated from the fault tree in (Fig. 4) (SFT_Formula.v).



```
//typedef enum {Normal, Waiting, Trip} SFT_state;
//SFT_State is corresponding to {0,1,2} of Current_State and Previous_State, for convenience sake.

`define k_Hys 10
//`define k_Trip_Delay 20


//SFT_Formula
module Property_Formula(f_X, f_X_Valid, f_Module_Error, f_Channel_Error, h_X_Setpoint, Current_State, Previous_State, th_X_Trip);

input[0:6] f_X;
input f_X_Valid;
input f_Module_Error;
input f_Channel_Error;
input[0:6] h_X_Setpoint;
input[0:1] Current_State;
input[0:1] Previous_State;

output th_X_Trip;

assign th_X_Trip =
//Fairncess
   ((f_X_Valid == 1 || f_Module_Error == 1 || f_Channel_Error == 1) && Current_State != 2)?0:          // (F1)
   ((f_X_Valid == 0 && f_Module_Error == 0 && f_Channel_Error == 0) && Current_State == 2)?1:    // (F2)
   ((f_X >= h_X_Setpoint) && Current_State != 2)?0:                                                 // (F3)
   ((f_X < h_X_Setpoint - `k_Hys) && Current_State == 2)?1:                                         // (F4)

//Correctness
   (Current_State == 0 && Previous_State != 0)?1:          // (C1)
   (Current_State == 1 && Previous_State != 1)?1:          // (C2)
   (Current_State == 2 && Previous_State == 2)?0:          // (C3)
   (Current_State == 2 && Previous_State != 2)?0:1;        // (C3)

endmodule
```

Fig. 7. A Verilog program translated from safety properties (Property_Formula.v).



```
JUNBEOM YOO@JUNBEOMYOO-PC ~/vis/vis-2.0/SFTA_Example
$ vl2mv SFT_Formula.v
SFT_Formula.v

JUNBEOM YOO@JUNBEOMYOO-PC ~/vis/vis-2.0/SFTA_Example
$ vl2mv Quality_Formula.v
Property_Formula.v

JUNBEOM YOO@JUNBEOMYOO-PC ~/vis/vis-2.0/SFTA_Example
$ ../vis.exe
vis release 2.0 (compiled Sat Jun 14 12:02:36     2008)
vis> read_blif_mv SFT_Formula.mv
vis> flatten_hierarchy
vis> static_order
vis> build_partition_mdds
vis> comb_verify Property_Formula.mv
th_X_Trip from network1 and th_X_Trip from network2 differ on input values
Current_State<0>:1
Current_State<1>:0
Previous_State<0>:0
Previous_State<1>:0
f_Channel_Error:1
f_Module_Error:1
f_X<0>:1
f_X<1>:1
f_X<2>:0
f_X<3>:0
f_X<4>:1
f_X<5>:1
f_X<6>:0
f_X_Valid:1
h_X_Setpoint<0>:1
h_X_Setpoint<1>:1
h_X_Setpoint<2>:1
h_X_Setpoint<3>:1
h_X_Setpoint<4>:1
h_X_Setpoint<5>:1
h_X_Setpoint<6>:1
Networks are NOT combinationally equivalent.

vis> quit
JUNBEOM YOO@JUNBEOMYOO-PC ~/vis/vis-2.0/SFTA_Example
$
```

Fig. 8. A verification process of combinational equivalence checking using VIS.

a counterexample in tabular notation using integer values. It can also visualize the result in a flowchart-like manner to help domain experts better understand the result. Such a feature is especially useful when analyzing the results of sequential equivalence checking and model checking. Results are often too complicated and long to understand. Every time we execute the combinational equivalence checking, VIS presents different counterexamples if possible.

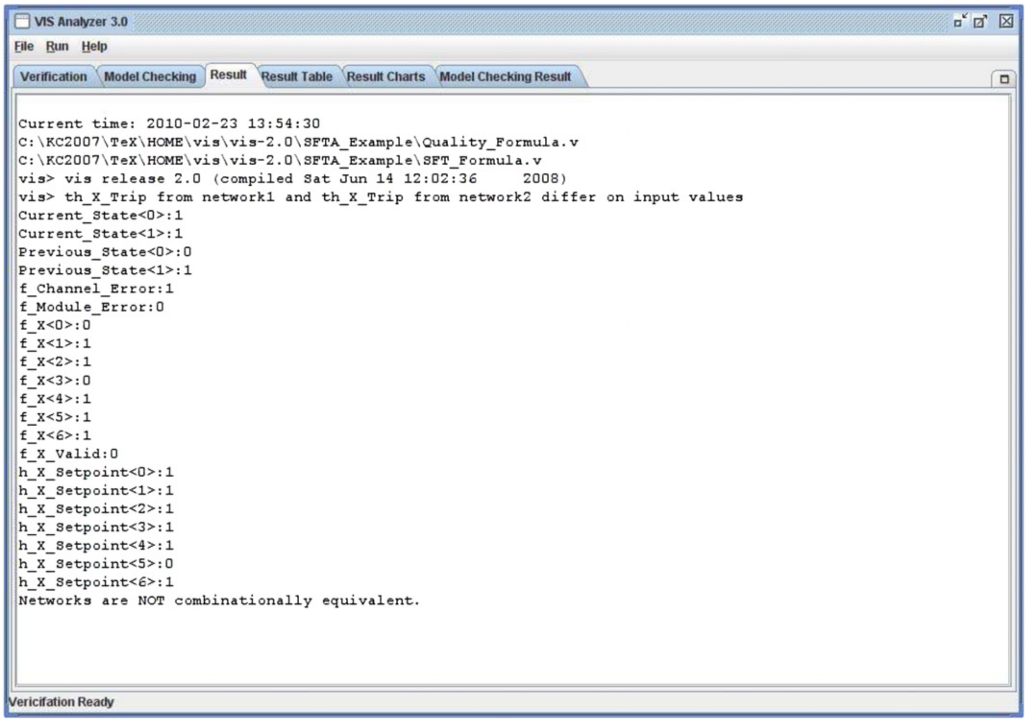
4.3. Analysis of verification results

The counterexample presented in (Fig. 8) helped engineers and verification experts to accurately understand the cause of the difference between the two models. In the requirements specification depicted in (Fig. 2), the counterexample is interpreted as follows: "In '*Waiting*' state, three input errors occur ($f\_X\_Valid = 1$, $f\_Channel\_Error = 1$ and $f\_Module\_Error = 1$) and the condition '$f\_X < h\_X\_Setpoint$' is satisfied at the same time, then the difference occurs". We analyzed the NuSCR specification for the *th_X_Trip* described in (Fig. 2(b)). In state '*Waiting*', any one of the three errors makes it transit to the state '*Trip*' and fire the shutdown signal ($th\_X\_Trip := 0$). If, however, the condition '$f\_X < h\_X\_Setpoint$' is satisfied, then it can also transit to the state '*Normal*'. Consequently, the state '*Waiting*' had two outgoing transitions, but can be selected nondeterministically. It was a logical error found in the NuSCR specification, since all constructs in NuSCR should be complete and consistent [11].

(Fig. 10) illustrates a corrected version of *th_X_Trip* in (Fig. 2(b)). Nondeterministic transitions occurred because the shutdown signal ($th\_X\_Trip := 0$) could be fired in two ways (i.e., '*Trip_By_Logic*' and '*Trip_By_Error*'). Different types of shutdown conditions should be distinguished explicitly as shown in (Fig. 10). When VIS's combinational equivalence checking was applied again on the modified definition of *th_X_Trip*, verification was successful in that "*Networks are combinationally equivalent*".

It must be emphasized that the error described above was NOT artificially seeded by the authors and that it was a genuine mistake made by professional engineers in the early state of system development. It is the same example used in our earlier papers (e.g., [5]). Using the procedure we explain in this paper, the logical mistake was found early, and the fix was properly made in subsequent versions of the NuSCR specification.

Because fault trees are usually generated during a relatively early state of requirements engineering, additional safety requirements are often derived. If one can utilize a fault tree as a verification aid, as we demonstrate in this paper, it is quite likely that


VIS Analyzer 3.0
File Run Help
Verification Model Checking Result Result Table Result Charts Model Checking Result
Vericifation Ready

```
Current time: 2010-02-23 13:54:30
C:\KC2007\TeX\HOME\vis\vis-2.0\SFTA_Example\Quality_Formula.v
C:\KC2007\TeX\HOME\vis\vis-2.0\SFTA_Example\SFT_Formula.v
vis> vis release 2.0 (compiled Sat Jun 14 12:02:36     2008)
vis> th_X_Trip from network1 and th_X_Trip from network2 differ on input values
Current_State<0>:1
Current_State<1>:1
Previous_State<0>:0
Previous_State<1>:1
f_Channel_Error:1
f_Module_Error:0
f_X<0>:0
f_X<1>:1
f_X<2>:1
f_X<3>:0
f_X<4>:1
f_X<5>:1
f_X<6>:1
f_X_Valid:0
h_X_Setpoint<0>:1
h_X_Setpoint<1>:1
h_X_Setpoint<2>:1
h_X_Setpoint<3>:1
h_X_Setpoint<4>:1
h_X_Setpoint<5>:0
h_X_Setpoint<6>:1
Networks are NOT combinationally equivalent.
```

Fig. 9. Verification results produced by the VIS Analyzer 3.0.

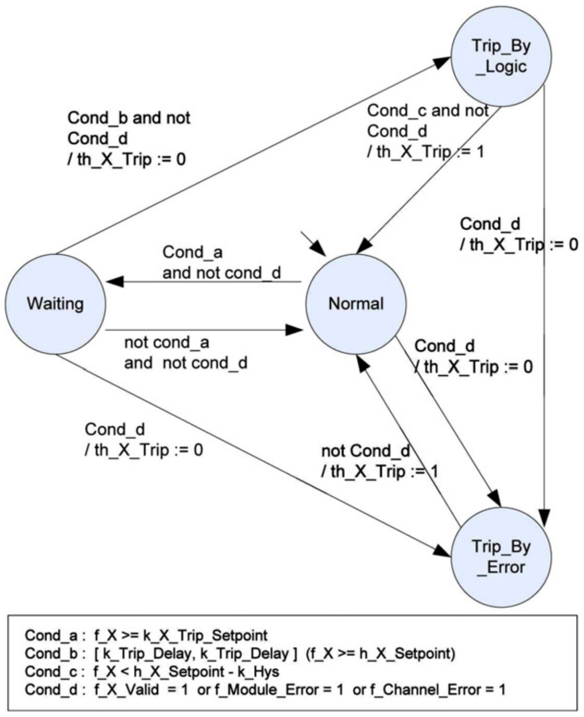

Trip_By _Logic
Cond_b and not Cond_d / th_X_Trip := 0
Cond_c and not Cond_d / th_X_Trip := 1
Cond_d / th_X_Trip := 0
Cond_a and not cond_d
Waiting
Normal
not cond_a and not cond_d
Cond_d / th_X_Trip := 0
Cond_d / th_X_Trip := 0
not Cond_d / th_X_Trip := 1
Trip_By _Error
Cond_a : f_X >= k_X_Trip_Setpoint
Cond_b : [k_Trip_Delay, k_Trip_Delay] (f_X >= h_X_Setpoint)
Cond_c : f_X < h_X_Setpoint - k_Hys
Cond_d : f_X_Valid = 1 or f_Module_Error = 1 or f_Channel_Error = 1

Fig. 10. A modified NuSCR definition of *th_X_Trip*.

potential semantic errors are detected early prior to the initiation of full-scale verification. Therefore, one can further increase trust in software and reduce the cost necessary for software development.

5. Practical issues

While useful, there are several practical issues to address for this technique to become more useful in industry. They are: (1) understandability of the constructed fault trees; (2) optimization of the logic formula; (3) mechanization of the translation from fault tree logics into Verilog programs; (4) combinational vs. sequential equivalence checking; (5) compositional fault tree analysis; and (6) an evaluation study.

Understandability of the constructed fault trees: When working on a complex system, a fault tree often contains too many nodes (e.g., several hundred nodes or more) for safety engineers to effectively understand. The fault tree synthesis algorithm needs to be enhanced to reduced potential redundancies in the fault trees. To accomplish the objective, we plan to work on defining semantics used in the NuFTA more concisely. Utilization of ideas developed in the model checking research community would prove useful in better managing the complexity when conducting backward analysis. In addition, backward analysis of real-time behavior needs to be fully supported.

Optimization of the logic formula: Brute-force application of minimal cut-set analysis apparently does not scale well in industrial projects. For example, the logic formula described in (Fig. 3) represents the minimal cut-set of the trip signal in the *LO_SG1_LEVEL* trip logic, but it has too many Boolean combinators. We plan to enhance NuFTA to generate more concise logic formulae through unfolding and optimizing techniques. Current implementation assigns each value of the process monitor variable into each leaf node. While intuitive, it is not necessarily an efficient strategy. Furthermore, propositional logic used in the current implementation is not powerful enough to support all features included in the NuSCR specification. We plan to revisit the issue of fault tree semantic definition.

Automated translation of fault trees into Verilog: The proposed verification technique constructs a software fault tree mechanically from NuSCR formal requirement specifications and then translates it into a logic formula with the support of NuFTA. When engineers translate the formula into Verilog programs, errors might occur in the process. Both logic formulae produced by NuFTA and Verilog programs are mathematical functions. If internal information on state history need not be tracked, one can build an automated translator based on our earlier work [56,59]. Safety properties, often available only in natural language format, must also be translated first into a logic formula and then into a Verilog program in hand. While the first step can't be automated, we plan to develop a translator for the second step, after refining the semantics of a software fault tree constructed from the NuSCR specification firmly.

Combinational vs. sequential equivalence checking: Combinational checking is a quick mathematical comparison between two functions which have no internal states to keep information. Otherwise, we must apply sequential equivalence checking, and it is slower. The current version of NuFTA produces a fault tree and a logic formula, which are corresponding to just one execution cycle of the whole system.

Compositional fault tree analysis: Fault trees and logic formulae produced by NuFTA are often too complicated and large to be used as inputs to VIS equivalence checking. While the example used throughout the paper is the fault tree against one node *th_X_Trip* in the NuSCR specification Fig. 2(b), in practice, all nodes in the FOD, described in (Fig. 3), should be used to construct a fault tree for one trip/pretrip signal.

An evaluation study: Results to date on extending the use of fault trees beyond that of safety analysis appear promising. Fault trees can guide an early application of safety-focused model checking. To determine applicability of the idea to real-world applications, we plan to conduct a fuller and more realistic experiment in which larger and more complex examples in RPS design are used. To conduct more industrial case studies, we must first develop a translator from fault trees into Verilog programs.

6. Conclusion

This paper proposed a safety-focused verification technique where a software fault tree mechanically constructed from formal specifications is used as a formal verification tool. This is possible because a fault tree is an abstract model of its software specification or source code, containing information relevant to the root-node event. We first translate the fault tree into a logic formula, and then into a Verilog program which is an input front-end of the VIS verification system. Properties which the system should preserve are also abstracted from aspects of the fault tree constructed and translated into a Verilog program too. It then performs the VIS's combinational equivalence checking to check whether the model of the fault tree satisfies the model of requirements properties.

Combinational equivalence checking is more cost-effective than full-scale verification methods such as sequential equivalence checking and model checking. We used a prototype version of the KNICS RPS in Korean nuclear power plants to demonstrate its effectiveness, and it showed that the proposed technique can detect important safety errors in requirements specification early.

References

[1] W.E. Vesely, F.F. Goldberg, N.H. Roberts, D.F. Haasl, Fault tree handbook, Technical report NUREG-0492, US Nuclear Regulatory Commission (1981).
[2] N.G. Leveson, SAFEWARE, System Safety and Computers, Addison Wesley, 1995.
[3] K. Vemuri, J. Dugan, K. Sullivan, Automatic synthesis of fault trees for computer-based systems, IEEE Transactions on Reliability 48 (4) (1999) 394–402.
[4] Y. Papadopoulos, J. McDermid, R. Sasse, G. Heiner, Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure, Reliability Engineering and System Safety 71 (3) (2001) 229–247.
[5] T. Kim, J. Yoo, S. Cha, A synthesis method of software fault tree from NuSCR formal specification using templates, Journal of the Korean Institute of Information Scientists and Engineers-Software and Application 32 (12) (2005) 1178–1191 (in Korean).
[6] J. Yoo, S. Cha, H.S. Son, Automatic generation of goal-tree from statecharts requirements specification, America Nuclear Society Transactions 88 (2003) 37–38.
[7] N.G. Leveson, P. Harvey, Analyzing software safety, IEEE Software 9 (5) (1983) 569–579.
[8] Y. Oh, J. Yoo, S. Cha, H.S. Son, Software safety analysis of function block diagrams using fault trees, Reliability Engineering and System Safety 88 (3) (2005) 215–228.
[9] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S.A. Edwards, S.P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, T. Villa, VIS: a system for verification and synthesis. in: Eighth International Conference on Computer Aided Verification, CAV'96, 1996, pp. 428–432.
[10] J. Yoo, S. Cha, C.H. Kim, Y. Oh, Formal software requirements specification for digital reactor protection systems, Journal of the Korean Institute of Information Scientists and Engineers-Software and Application 31 (6) (2004) 750–759 (in Korean).
[11] J. Yoo, T. Kim, S. Cha, J.-S. Lee, H.S. Son, A formal software requirements specification method for digital nuclear plants protection systems, Journal of Systems and Software 74 (1) (2005) 73–83.
[12] R. Manian, J. Dugan, D. Coppit, K. Sulliva, Combining various solution techniques for dynamic fault tree analysis of computer systems, in: 3rd IEEE International High-Assurance Systems Engineering Symposium, 1998, pp. 21–28.
[13] K. Sulliva, J. Dugan, D. Coppit, The Galileo fault tree analysis tool, in: IEEE International Symposium of Fault Tolerant Computing, FTC-29, 1999, pp. 232–235.
[14] G. Palshikar, Temporal fault trees, Information and Software Technology 44 (3) (2002) 137–150.
[15] M. Walker, Y. Papadopoulos, Qualitative temporal analysis: Towards a full implementation of the fault tree handbook, Control Engineering Practice 17 (10) (2009) 1115–1125.
[16] G. Bruns, S. Anderson, Validating safety models with fault trees, in: 12th International Conference on Computer Safety, Reliability, and Security, 1993, pp. 21–30.
[17] K. Hansen, A. Ravn, From safety analysis to software requirements, IEEE Transactions of Software Engineering 24 (7) (1998) 573–584.
[18] G. Schellhorn, A. Thums, W. Reif, Formal fault tree semantics, in: The 6th World Conference on Integrated Design and Process Technology (IDPT-2002), 2002.
[19] D. Coppit, K. Sullivan, J. Dugan, Formal semantics of models for computational engineering: a case study on dynamic fault trees, in: the 11th International Symposium on Software Reliability Engineering, ISSRE-2000, 2000, pp. 270–282.
[20] J. Xiang, K. Ogata, W. Kong, K. Futatsugi, From fault tree analysis to formal system specification and verification with ots/cafeobj, Computer Software-JSSST Journal 23 (3) (2006) 573–584.
[21] F. Ortmeier, G. Schellhorn, Formal fault tree analysis: practical experiences, Electronic Notes in Theoretical Computer Science 185 (17) (2007) 139–151.
[22] A. Thums, Formale fehlerbaumanalyse, Ph.D. Thesis, Universit at Augsburg, 2004 (in German).
[23] P. Fenelon, J. McDermid, An integrated toolset for software safety analysis, Journal of Systems and Software 21 (3) (1993) 279–290.
[24] B. Kaiser, P. Liggesmeyer, O. Mackel, A new component concept for fault trees, in: The 8th Australian Workshop on Safety Critical Systems and Software, SCS 2003, 2003, pp. 37–46.
[25] L. Grunske, B. Kaiser, Automatic generation of analyzable failure propagation models from component-level failure annotations, in: The 5th International Conference on Quality Software, QSIC 2005, 2005, pp. 117–123.
[26] L. Grunske, B. Kaiser, Y. Papadopoulos, Model-driven safety evaluation with state-event-based component failure annotations, in: 8th International Symposium on Component-Based Software Engineering, CBSE 2005, 2005, pp. 33–48.
[27] B. Kaiser, C. Gramlich, M. Forster, State/event fault trees. A safety analysis model for software-controlled systems, Reliability Engineering and System Safety 92 (11) (2007) 1521–1537.
[28] N.G. Leveson, S. Cha, T. Shimeall, Safety verification of ada programs using software fault trees, IEEE Software 8 (4) (1991) 48–59.
[29] S.-Y. Min, Y. kyu Jang, S. Cha, Y.-R. Kwon, D. Bae, Safety verification of ada95 programs using software fault trees, in: Computer Safety, Reliability and Security (SAFECOMP), in: LNCS, vol. 1698/1999, 1999, pp. 226–238.
[30] G.-Y. Park, K.Y. Koh, E. Jee, P.H. Seong, Fault tree analysis of KNICS RPS software, Nuclear Engineering and Technology 40 (5) (2008) 397–408.
[31] I.E. Commission, International Standard for Programmable Controllers: Programming Languages, Part 3, 1993.
[32] R. Mojdehbakhsh, S. Subramanian, R. Vishnuvajjala, W. Tsai, L. Elliott, A process for software requirements safety analysis, in: International Symposium on Software Reliability Engineering, 1994 pp. 45–54.
[33] V. Ratan, K. Partridge, J. Reese, N. Leveson, Safety analysis tools for requirements specifications, in: The 7th COMPASS Workshop, 1996, pp. 149–160.
[34] D. Harel, On visual formalism, Communication of the ACM 31 (5) (1986) 514–530.
[35] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese, Requirements specification for process-control systems, IEEE Transactions on Software Engineering 20 (9) (1994) 684–707.
[36] Y. Papadopoulos, M. Maruhn, Model-based synthesis of fault trees from matlab-simulink models, in: The 2001 International Conference on Dependable Systems and Networks, DSN '01, 2001, pp. 77–82.
[37] K. Sullivan, J. Dugan, C. Coppit, The Galileo fault tree analysis tool, in: The 29th Annual International Symposium on Fault-Tolerant Computing, 1999, pp. 232–235.
[38] UPPAAL, http://www.uppaal.com/ (2010).

[39] C.L. Heitmeyer, R.D. Jeffords, B.G. Labaw, Automated consistency checking of requirements specifications, IEEE Transactions on Software Engineering 5 (3) (1996) 231–261.
[40] K. Jensen, L. Kristensen, Coloured Petri Nets, in: Modelling and Validation of Concurrent Systems, Springer, 2009.
[41] J.M. Spivey, The Z Notation: A Reference Manual, 2nd ed., Prentice Hall, 1992.
[42] S. Mishra, D. Kushwaha, A. Misra, Hybrid reliable load balancing with mosix as middleware and its formal verification using process algebra, Future Generation Computer System, in press (doi:10.1016/j.future.2010.12.007).
[43] S. Specification, D.L.F. Society, http://www.sdl-forum.org/.
[44] M. Huth, M. Ryan, in: Logic in Computer Science, 2nd ed., Cambridge, 2004.
[45] PVS: Specification and Verification System, http://pvs.csl.sri.com/.
[46] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.
[47] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (2) (1986) 244–263.
[48] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.
[49] S.-Y. Huang, K.-T. Cheng, Formal Equivalence Checking and Debugging, Kluwer Academic Publishers, 1998, (Chapter 4).
[50] SPIN, http://spinroot.com/.
[51] CBMC: Bounded Model Checking for ANSI-C, http://www.cprover.org/cbmc/.
[52] BLAST: Berkeley Lazy Abstraction Software Verification Tool, http://mtc.epfl.ch/software-tools/blast/.
[53] J. Yoo, E. Jee, S.S. Cha, Formal modeling and verification of safety-critical software, IEEE Software 26 (3) (2009) 42–49.
[54] S. Jung, J. Yoo, S. Cha, VIS analyzer: a visual assistant for vis verification and analysis, in: The 13th IEEE Computer Society symposium dealing with the rapidly expanding field of object/component/service-oriented real-time distributed computing (ORC) technology, ISORC 2010 Symposium, 2010.
[55] G.-Y. Park, J.-S. Lee, S.-W. Cheon, K.-C. Kwon, E. Jee, K.Y. Koh, Safety analysis of safety-critical software for nuclear digital protection system, in: Computer Safety, Reliability and Security (SAFECOMP), in: LNCS, vol. 4680/2007, 2007.
[56] J. Yoo, E. Jee, S. Cha, A verification framework for FBD based software in nuclear power plants, in: The 15th Asia Pacific Software Engineering Conference (APSEC), 2008, pp. 385–392.
[57] T. Henzinger, Z. Manna, A. Pnueli, Timed transition systems, REX Workshop, 1991, pp. 226–251.
[58] S. Yun, D.-A. Lee, J. Yoo, NuFTA: a case tool for automatic software fault tree analysis, in: Transactions of the Korean Nuclear Society Spring Meeting 2010, 2010.
[59] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, P. Seong, FBD verifier: interactive and visual analysis of counterexample in formal verification of function block diagram, Journal of Research and Practice in Information Technology 42 (3) (2010) 255–272.

Sungdeok Cha is a professor in Korea University's Computer Science and Engineering Department. His research interests include software safety and computer security. Cha has a Ph.D. in information and computer science from the University of California, Irvine. Contact him at scha@korea.ac.kr.

Junbeom Yoo is an assistant professor in Konkuk University's Department of Computer Science and Engineering. His research interests include formal methods and safety analysis. Yoo has a Ph.D. in computer science from the Korea Advanced Institute of Science and Technology. Contact him at jbyoo@konkuk.ac.kr.