Available online at www.sciencedirect.com
SCIENCE DIRECT®
Reliability Engineering and System Safety 82 (2003) 11–20
RELIABILITY ENGINEERING & SYSTEM SAFETY
www.elsevier.com/locate/ress

Systematic evaluation of fault trees using real-time model checker UPPAAL

Sungdeok Cha[a], Hanseong Son[b], Junbeom Yoo[a], Eunkyung Jee[a,*], Poong Hyun Seong[c]

[a]*Division of Computer Science, EECS Department and Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST), 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, South Korea*
[b]*MMIS Team, Korea Atomic Energy Research Institute (KAERI), Taejon 305-353, South Korea*
[c]*Department of Nuclear and Quantum Engineering, Korea Advanced Institute of Science and Technology (KAIST), Taejon 305-701, South Korea*

Received 16 November 2002; accepted 14 February 2003

Abstract

Fault tree analysis, the most widely used safety analysis technique in industry, is often applied manually. Although techniques such as cutset analysis or probabilistic analysis can be applied on the fault tree to derive further insights, they are inadequate in locating flaws when failure modes in fault tree nodes are incorrectly identified or when causal relationships among failure modes are inaccurately specified. In this paper, we demonstrate that model checking technique is a powerful tool that can formally validate the accuracy of fault trees. We used a real-time model checker UPPAAL because the system we used as the case study, nuclear power emergency shutdown software named Wolsong SDS2, has real-time requirements. By translating functional requirements written in SCR-style tabular notation into timed automata, two types of properties were verified: (1) if failure mode described in a fault tree node is consistent with the system's behavioral model; and (2) whether or not a fault tree node has been accurately decomposed. A group of domain engineers with detailed technical knowledge of Wolsong SDS2 and safety analysis techniques developed fault tree used in the case study. However, model checking technique detected subtle ambiguities present in the fault tree.

Keywords: Software engineering; Formal methods; Fault tree analysis; Model checking

1. Introduction

Fault tree analysis [1] is one of the most frequently applied safety analysis techniques [2] when developing safety-critical and often time-critical industrial systems. It is a goal-driven and backward analysis technique where analysis starts from the hazardous system state specification and credible causes leading to the top event and relationship among them are visually documented. Fault tree analysis attempts to convince the analyst that the system is free from encountering or contributing to the occurrence of the top-level event. One must possess, in addition to general understanding of safety analysis techniques, detailed and domain-specific knowledge to perform fault tree analysis. Although widely used in industry, fault tree analysis has fundamental limitation that it is informal in nature [3]. Graphical notations help analyst organize thought process systematically, but the technique itself offers no help in investigating causal events and the relationship among them. Therefore, when different experts apply fault tree analysis, result is not guaranteed to be repeatable, and analysis may contain flaws. Inspection technique [4] can be used to detect errors of fault trees, but it, too, is informal in nature. Other techniques [13,14], such as variants of Monte Carlo simulation, Birnbaum's importance measure, and Fussell–Vesely measure also can be applied to validate fault trees. While these approaches are useful in deriving quantitative conclusion such as the probability of a system failing in a specific manner, they are inadequate in determining if all credible failure modes are captured or causal relationship among causal events are accurately identified.

* Corresponding author.
E-mail addresses: cha@salmosa.kaist.ac.kr (S. Cha); hsson@kaeri.re.kr (H. Son); jbyoo@salmosa.kaist.ac.kr (J. Yoo); ekjee@salmosa.kaist.ac.kr (E. Jee); phseong@mail.kaist.ac.kr (P.H. Seong).

0951-8320/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0951-8320(03)00059-0

Our research goal is to provide formal, automated and qualitative assistance to informal and/or quantitative safety analysis. In particular, we chose to validate the correctness of fault trees using a model checker because safety demonstration will always likely to be required by government authorities when granting licenses to operate safety-critical systems. Model checking [5] is a proven-effective and automated technique in verifying complex behavior of concurrent systems. A model checker, given the system description and property specification, determines if the properties hold in the model or not. Behavioral model is usually written in finite state machine, and the property specification is written in temporal logic. If the behavior of given system is infinite, users need to apply techniques such as abstraction to generate a model whose behavior is finite. For example, if a reactor temperature is represented in real number and the threshold values related to the trip conditions are defined, finite encoding of the temperature would include only three possible values (e.g. low, normal and high). Finiteness in behavior guarantees that model checking process, fully automated, will be terminated. Users need not worry about complex internals of model checking algorithms and data structures, and a model checker explores all possible reachable states before concluding that the property holds. Otherwise, a counterexample providing insight into debugging the model is generated. Such exhaustive search is a powerful tool to safety engineers who are practically unable to investigate all possible paths and states. There are several model checkers being used in industry, and SMV [6] and SPIN [7] are widely used in industry. Unfortunately, they do not support verification of time-related system behavior. In this paper, we use a real-time model checker UPPAAL [8] to validate the correctness of fault trees because the system used as the case study has real-time constraints.

The remainder of this paper is as follows. Section 2 briefly introduces the real-time model checker UPPAAL. In Section 3, we describe the experimental safety analysis performed on Wolsong SDS2 system. Section 4 shows the evaluation of fault tree using model check UPPAAL. Conclusion and future work direction are discussed in Section 5.

2. Real-time model checking using UPPAAL

In our approach, we use a real-time model checker UPPAAL [8] to validate the correctness of fault trees. The toolset, shown in Fig. 1, consists of an editor,

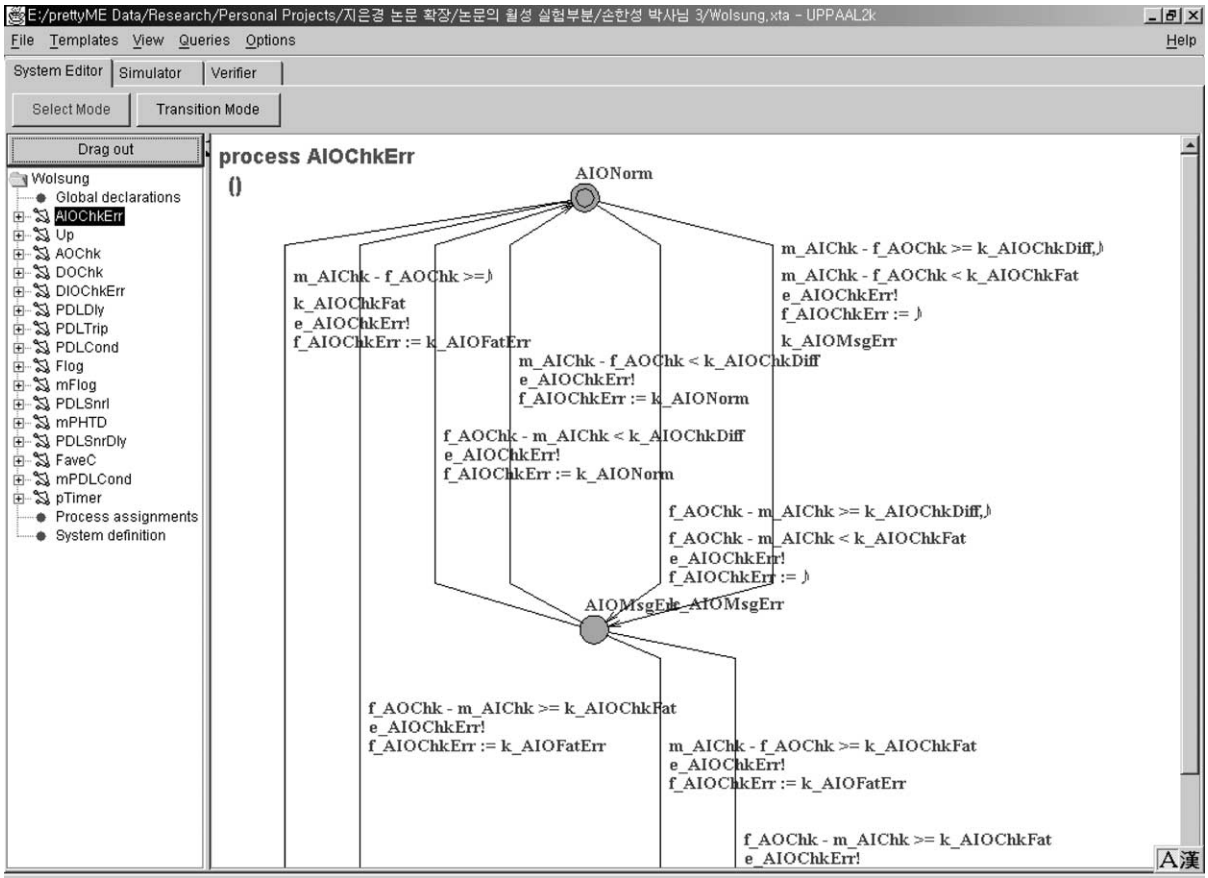

Fig. 1. UPPAAL toolset.

simulator, and verifier, and it has been used to analyze behavior of protocol specification, control boxes, and algorithms. Editor allows users to develop or revise a set of non-deterministic processes extended with clock and data variables.

Behavioral model used by UPPAAL model is the timed automata, developed by Alur and Dill [15], which extends classical finite state automata with clock variables. Current implementation of UPPAAL supports a system model that consists of a collection of timed automata extended with integer (data) variables in addition to clock variables. Automata may communicate with each other via shared integer variables or using communication channels. The UPPAAL model checker is designed to check for simple invariant, reachability, and bounded reachability properties. UPPAAL's property specifications are based on CTL (Computational Tree Logic) [16]. UPPAAL accepts the property specification of the following format:

$$\varphi :: = \forall\Box\beta|\exists\Diamond\beta$$

$$\beta :: = a|\neg\beta|(\beta)|\beta\vee\beta|\beta\wedge\beta|\beta\rightarrow\beta$$

where a represents a clock variable, data variable expression, or location in the timed automata. An atomic expression represents integer ranges (e.g. $1 \leq x \leq 5$)—not real numbers—on the clock or data variables or difference between two variables (e.g. $3 \leq x - y \leq 7$). The property specification $\forall\Box\beta$ states that the formula β is to be satisfied in all states reachable from the initial state. Similarly, $\exists\Diamond\beta$ is true if the formula β is eventually satisfied in one or more reachable states.[1]

Although property specification accepted by UPPAAL can be an arbitrarily complex temporal logic formula [9], we found the following property patterns to be particularly useful when validating the correctness of fault trees.

– $\forall\Box(\neg p_N)$: Let p_N be the temporal logic formula semantically equivalent to the failure mode described in the fault tree node N. The property $\forall\Box(\neg p_N)$ determines if system can ever reach such state. If model checker returns TRUE, the state denoted by p_N will never occur, and the system is free from such hazard. It means that a safety engineer has thought a logically impossible event to be feasible, and the model checker found an error in the fault tree. If, on the other hand, the property is not satisfied, such failure mode is indeed possible, and the model checker generates detailed (but partial) information on how such hazard may occur. Detailed analysis of the counterexample may provide assurance that safety analysis has been properly applied. Another possibility is that the counterexample may reveal a failure mode which human expert had failed to consider.

– $\forall\Box((B_1 \wedge \cdots \wedge B_n) \rightarrow A)/\forall\Box((B_1 \vee \cdots \vee B_n) \rightarrow A)$: This pattern is used to validate if AND/OR connectors, used to model relationship among causal events, are correct. One can conclude that the refinement of fault tree was done properly if model checker returns true. Otherwise, there are two possibilities: (1) gate connector is incorrect; or (2) failure modes in the lower level fault tree nodes are incorrect. A counterexample can provide insight as to why the verification failed and how the fault tree might be corrected.

Unfortunately, it is impossible to fully validate the correctness of fault trees using counterexamples alone because model checkers provide just one scenario where the property is not satisfied. While such information can certainly provide useful insights to safety engineers why the property did not hold, model checkers are not designed to provide complete and exhaustive list of counterexamples.

3. Safety analysis on Wolsong SDS2

Wolsong SDS2 [10–12] is a software-implemented shutdown system and has been in service[2] in Korea for the past six years. It is required to continually monitor the state of the power plant by reading various sensor inputs (e.g. reactor temperature and pressure) and generating a trip signal should the reactor is found to be in an unsafe state. Among the six trip parameters, we have used the Primary Heat Transport Low Core Differential Pressure (PDL) trip condition as an example because it is the most complex trip condition and has time-related requirements. Trip signal can be either an immediate trip or a delayed trip, and both trips can be simultaneously enabled. Delayed trip occurs if the system remains in certain states for over a period of time. High-level requirements for PDL trip were written in English in a document called the Program Functional Specification (PFS) as shown partially below:

"[…] If the D/I is open, select the 0.3%FP conditioning level. If $\phi_{\mathrm{LOG}} < 0.3\%\mathrm{FP} - 50$ mV, condition out the immediate trip. If $\phi_{\mathrm{LOG}} \geq 0.3\%\mathrm{FP}$, enable the trip.

[1] Since characters used to represent logical quantifiers are missing in the standard keyboard, UPPAAL uses 'A[ ]' and '⟨E⟩' to denote '∀□' and '∃◇', respectively. For example, E⟨ ⟩ (p1.cs and p2.cs): It is possible for two processes, p1 and p2, to be in the critical section simultaneously; A[ ] (c2 > 30 imply a = =0): Whenever c2 is greater than 30, value of data variable a must always be zero.

[2] Software and supporting documents, including requirements specification, design specification, and safety analysis reports, were subject to inspection when granting an operational license and no formal methods were applied.

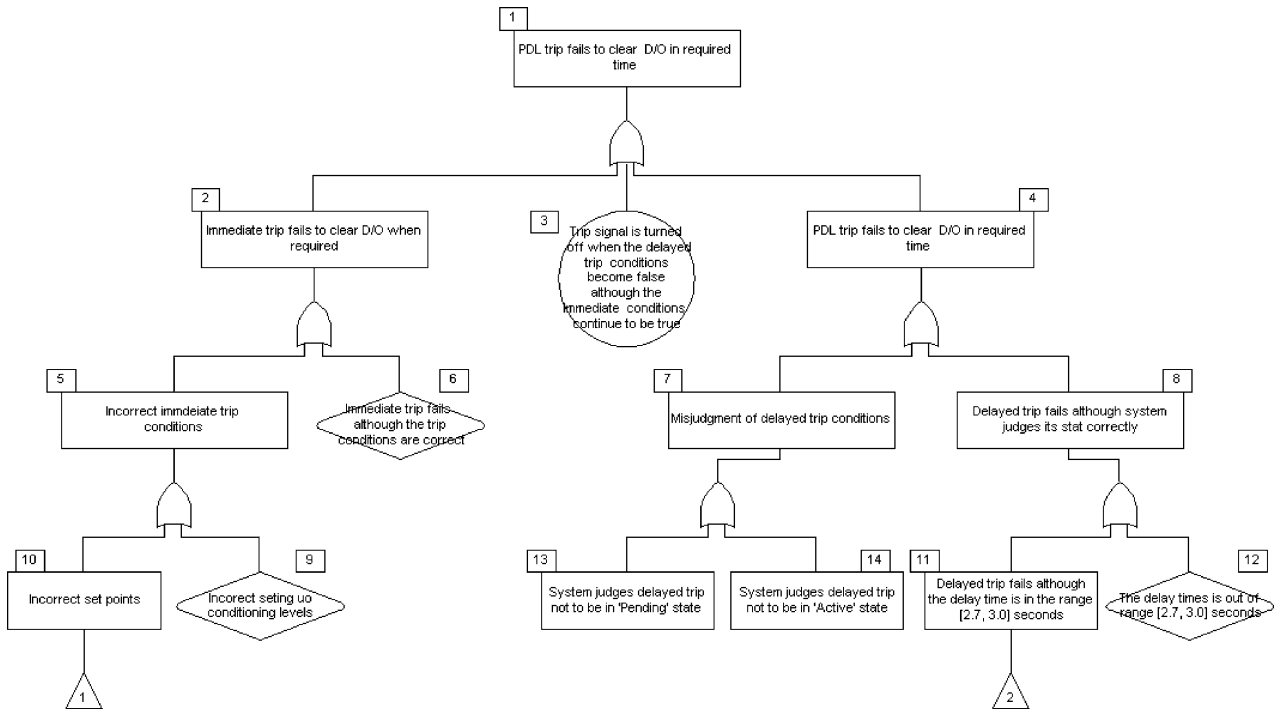

Fig. 2. A part of fault tree of Wolsong PDLTrip.

Annunciate the immediate trip conditioning status via the PHT ΔP trip inhibited ($\phi_{LOG} < 0.3\%$FP) window D/O.
[…]
If any ΔP signal is below the delayed trip setpoint and f_{AVEC} exceeds 70%FP, open the appropriate loop trip error message D/O. If no PHT DP delayed trip is pending or active, then execute a delayed trip as follows:

a. Continue normal operation without opening the parameter trip D/O for normally three seconds. The exact delay must be in the range [2.7 s, 3.0 s].
b. Once the delayed parameter trip has occur, keep the parameter trip D/O open for one second(± 0.1 s), and then close the parameter trip D/O once all DP signals are above the delayed trip setpoint or f_{AVEC} is below 70% FP.

[…]"

When performing fault tree analysis, additional documents including a software requirements specification and software design documentation were used. These documents provide detailed and technical insight about the system, and they were thoroughly reviewed by a group of technical experts and government regulators before the operating license was granted. Fault tree, shown in Fig. 2, was initially developed by a group of graduate students majoring in software engineering who had previously reviewed Wolsong SDS2 documents and performed independent safety analysis. They are also familiar with technical knowledge of software safety in general and fault tree analysis in particular. Therefore, they possessed in-depth knowledge on how the trip conditions work. In addition, the fault tree was subsequently reviewed and revised by a group of domain experts in nuclear engineering who concluded that the fault tree appeared to be correct to the best of their knowledge.

The top-level event, derived from the results of preliminary hazard analysis, is given as 'PDL trip fails to clear digital output (D/O) in required time'. The fault tree node had been refined into three causal events connected by OR gate. Failure modes described in some nodes (e.g. 2 and 4) were further refined.

Validation of fault tree consists of the following steps:

(1) Translate structured decision tables into a set of concurrent timed automata.[3] Variables used in the timed automata follow convention used in four variable approach, and prefixes m_, c_, and k_ represent monitored variables, controlled variables, and constant values, respectively. For example, functional requirement "If $\phi_{LOG} < 0.3\%$FP $- 50$ mV, condition out the immediate trip" is captured by the rightmost transition

[3] Our UPPAAL model for PDLTrip in Wolsong SDS2 is described in Appendix A.

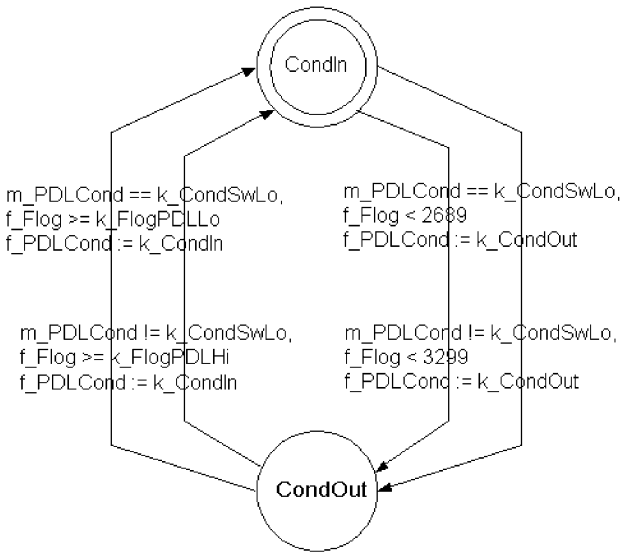

Fig. 3. Timed automata[5] for PDLCond trip condition.

of Fig. 3 (timed automata given in the figure can be represented in the text format as shown below:

```
process PDLCond{
  state  CondIn, CondOut;
  init   CondIn;
  trans CondIn → CondOut {
                    guard m_PDLCond = = k_CondSwLo, f_Flog < 2689;
                    assign f_PDLCond := k_CondOut;},
        CondOut → CondIn {
                    guard m_PDLCond = = k_CondSwLo, f_Flog > = 2739;
                    assign f_PDLCond := k_CondIn;},
        CondIn → CondOut {
                    guard m_PDLCond = = k_CondSwHi, f_Flog <3299;
                    assign f_PDLCond := k_CondOut;},
        CondOut → CondIn {
                    guard m_PDLCond = = k_CondSwHi, f_Flog > = 3349;
                    assign f_PDLCond := k_CondIn;}.)
```

labeled "If m_PDLCond = =k_CondSwLo and f_Flog < 2689, then f_PDLCond := k_CondOut.".[4] For the PDL trip alone, the complete specification consisted of 12 concurrent timed automata, and there were about 2^{15} feasible states, clearly too many to fully inspect manually;

(2) Derive properties to be verified using one of the two patterns described earlier; and
(3) Run UPPAAL to perform model checking.

4. Fault tree evaluation

4.1. Case 1: correctness of failure mode described in the node 3

Domain knowledge is needed to correctly rewrite the failure mode in temporal logic formula. In our example, the formula (f_PDLSnrI = =k_SnrTrip AND f_PDLCond = =k_CondIn) denotes the activation of immediate trip condition. Likewise, delayed trip is cancelled when the PDLDly process moves from the Waiting state to the Normal state and the value of f_PDLTrip becomes k_NonTrip in some states other than the initial state (e.g. denoted by having clock variable $z > 0$). Therefore, temporal logic formula corresponding to the absence of system state corresponding to the fault tree node 3, is given as follows:

$\forall\Box(\neg\, p_3)$ whereas p_3 corresponds to
(f_PDLSnrI = =k_SnrTrip and f_PDLCond = =k_CondIn) and
(f_PDLDly = =k_InDlyNorm and f_PDLTrip = =k_NotTrip and $z > 0$)

UPPAAL concluded that the property was not satisfied, and a counterexample, shown in terms of simulation trace, was generated as shown in Fig. 4. Each step can be replayed, and the tool graphically illustrates indicates which event took place in which configuration.[5] Simulation trace revealed that

[4] In the program functional requirements document, 0.3% of FP (full power) is defined as 2739 mV. Therefore, 0.3%FP − 50 mV evaluates to 2689 mV.

[5] Although detailed explanation of simulation trace shown in Fig. 4 is beyond the scope of this paper, the followings are worth brief explanation. The upper left window, 'Enabled Transitions', indicate all of the active transitions in the current system configuration. Window in the middle displays the values of data and clock variables in the current and visually highlighted configuration.

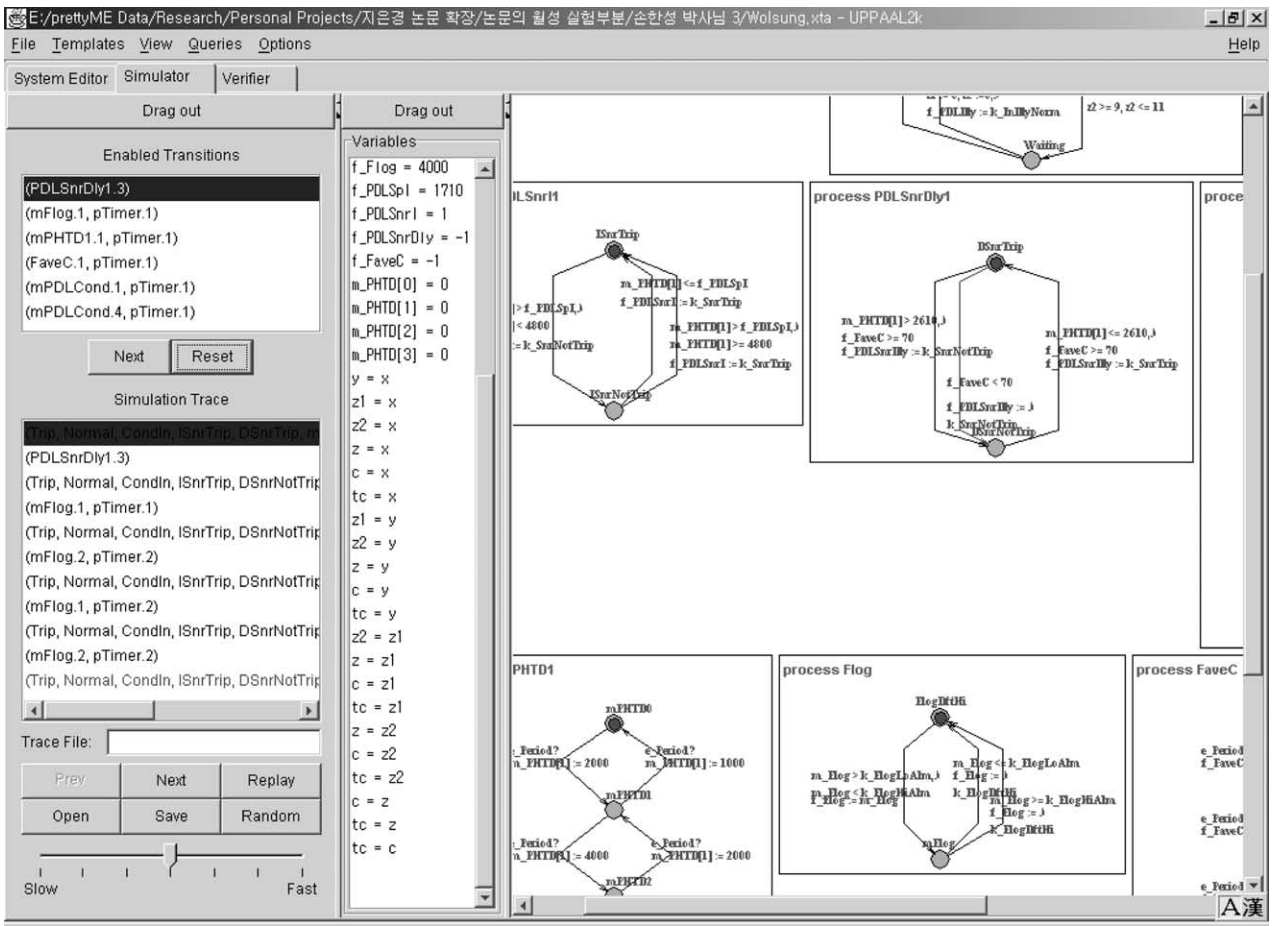

Fig. 4. Screen dump of the UPPAAL outputs.

the property did not hold if the trip signal is (incorrectly) turned off (e.g. becomes NotTrip) when the immediate trip condition becomes false while delayed trip condition continues to be true. This is possible because two types of trips have the same priority. While failure mode captured in the node 3 is technically correct when analyzed in isolation, model checking revealed that it was incomplete and that it must be changed to 'Trip signal is turned off 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.) Analysis of simulation trace provided safety analysts an interactive opportunity to investigate details of subtle failure modes human forgot to consider.

4.2. Case 2. Correctness of failure mode described in the node 12

Node 12 describes a failure mode where the system incorrectly clears a delayed trip signal outside the specified time range of [2.7 s, 3.0 s]. UPPAAL accepts only integers as the value of a clock variable, z in this example, and if we were to use 27 and 30 to indicate the required time zone, literal translation of the failure mode shown in the fault tree would correspond to the following:

$\forall\Box(\neg\, p_{12})$ where p_{12} is
(($z < 27$ or $z > 30$) and f_PDLTrip = =k_NotTrip)

However, model checking of this formula indicates that the property does not hold, and an analysis of the counterexample revealed that the predicate p_{12} does not hold when z is equal to zero (e.g. no time passed at all). This is obviously incorrect, based on domain-specific knowledge of how delayed trip is to work, and it quickly reminds a safety analyst that the failure mode, as it is written, is ambiguous in that the current description of the failure mode fails to explicitly mention that the system must be in the in the waiting state, not the initial system state, before the delayed trip timer is set to expire. That is, the property needs to be modified as follows:[6]

$\forall\Box(\neg\, p_{12})$ where p_{12} is
(f_PDLSnrDly = = k_SnrTrip and

[6] The following clause in the PFS provides clue as to how the formula is to be revised: "If any ΔP signal is below the delayed trip setpoint and f_{AVEC} exceeds 70% FP, open the appropriate loop trip error message D/O. If no PHT DP delayed trip is pending or active, then execute a delayed trip as follows: Continue normal operation without opening the parameter trip D/O for normally three seconds. The exact delay must be in the range [2.7 s, 3.0 s]."

f_FaveC > = 70) and ($z < 27$ or $z > 30$) and (f_PDLTrip = = k_NotTrip)

Model checking of the revised property demonstrated that the property is satisfied, and it means that the fault tree node 12 is essentially correct although it initially contained implicit assumptions. Application of model checking technique helped a safety engineer better understand the context in which specified failure mode occurs and therefore conduct a more precise safety analysis.

5. Conclusions and future work

This paper demonstrated that model checking technique is useful when we are to validate the correctness of informal safety analysis such as fault tree analysis. Whereas fault tree analysis largely depends on the domain-specific knowledge of human experts, it lacks formality and the process is not automated. By performing a straightforward transformation of the software requirements specification into timed automata, we were able to have a model checker, UPPAAL in our case study, examine all possible states exhaustively and determine if failure mode anticipated is indeed correct. We learned that model checking technique is a useful complement to fault tree analysis because it was able to identify subtle and previously omitted failure mode. In the case of the fault tree node (3, failure mode was correct by itself, but a safety engineer realized that the other case must also be considered when model checker failed to prove the property. When applied to fault tree node (12, model checker provided insight that the failure mode can be (and should be) revised to describe the identified failure mode more accurately.

It should be noted the group who developed that fault tree is familiar with safety techniques in general and the Wolsong SDS2 system in particular. In addition, fault tree was informally reviewed by a group of domain experts and they had concluded that the fault tree appeared to be correct. Results of model checking to validate the correctness of fault tree surprised technical staff involved in fault tree analysis and an independent review. Such experience convinced us that model checking technique is a useful, complementary, and automated tool to safety engineers. Even if model checking fails to identify previously undetected failure modes, one would have a strong confidence if a fault tree analysis result had been verified by a model checker.

We believe that formal methods can be more widely applied than reported in this paper in making sure that safety analyses have been performed correctly. For example, the proposed approach can validate correctness of the failure mode and effect analysis (FMEA) results. FMEA, popular among engineers developing safety-critical systems, depends on forward analysis and examines the impact of anticipated failure modes. Like fault tree analysis, FMEA is informal in nature. Correctness of FMEA can be formulated as a reachability problem using anticipated failure mode as the initial state, and model checker can examine if the state denoting the consequence of the failure mode is indeed reachable.

Acknowledgements

This work was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the multi-presence project at AITRC and Korea National Reserch Laboratory (NRL) Program.

Appendix A

This appendix describes the simulation trace as shown in Fig. A1, which is a result of the model checking for node 3 of the fault tree shown in Fig. 2.

The system is modeled as a tuple, (PDLTrip, PDLDly, PDLCond, PDLSnrI1, PDLSnrDly1, mFlog, mPHTD1, Flog, FaveC, mPDLCond, pTimer). For the convenience of the explanation, we gave numbers to each line of the trace. Each tuple on oddly numbered line shows a state of the system and tuples on evenly numbered line show the activated transitions at the state that the just above tuple represents. Followings are more detailed descriptions of this trace:

Line 1–3. These tuples represent that the delayed trip condition is disabled—due to unspecified environmental conditions—after both the immediate trip and delayed trip conditions are met. Note that the state of PDLSnrDly, which is one of the elements of the system tuple, changes from DSnrTrip to DSnrNotTrip through the transition (PDLSnrDly1.trans3) (line 2). Referring to Fig. A2 enables us to guess that this transition is happened because f_FaveC is less than 70 (f_FaveC < 70).

Line 3–7. The system reads the value of mPHTD, which is one of the inputs (line 4). This value changes the state of PDLSnrI, which determines the state of the immediate trip, from ISnrTrip(line 5) to ISnrNotTrip (line 7) by the transition (PDLSnrI1.trans3) (line 6). In other words, the value read is greater than the predefined trip set-point and thus the trip status turns to be normal. Refer to Fig. A3.

Line 7–9. The system reads the value of mPHTD again (line 8). This value makes the PDLSnrI state returned to the state of immediate trip. Fig. A4 shows the process of reading mPHTD.

Line 9–13. The system reads the value of FaveC twice (lines 10 and 12). Fig. A5 shows the process of reading FaveC.

Line 13–15. The FaveC value changes the state of PDLSnrDly, which determines the state of the delayed trip, from DsnrNotTrip to DsnrTrip by the transition (PDLSnrDly1.trans2) (line 14).

```
1: (Trip,Normal,CondIn,ISnrTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC0,CondSwLo,pTimer0)
2: (PDLSnrDly1.trans3)
3: (Trip,Normal,CondIn,ISnrTrip,DSnrNotTrip,mFlog0,mPHTD0,FlogDftHi,FaveC0,CondSwLo,pTimer0)
4: (mPHTD1.trans1, pTimer.trans1)
5: (Trip,Normal,CondIn,ISnrTrip,DSnrNotTrip,mFlog0,mPHTD1,FlogDftHi,FaveC0,CondSwLo,pTimer0)
6: (PDLSnrI1.trans3)
7:(Trip,Normal,CondIn,ISnrNotTrip,DSnrNotTrip,mFlog0,mPHTD1,FlogDftHi,FaveC0,CondSwLo,pTimer0)
8: (mPHTD1.trans4, pTimer.trans2)
9:(Trip,Normal,CondIn,ISnrNotTrip,DSnrNotTrip,mFlog0,mPHTD0,FlogDftHi,FaveC0,CondSwLo,pTimer0)
10: (FaveC.trans1, pTimer.trans2)
11:(Trip,Normal,CondIn,ISnrNotTrip,DSnrNotTrip,mFlog0,mPHTD0,FlogDftHi,FaveC1,CondSwLo,pTimer0)
12: (FaveC.trans3, pTimer.trans2)
13:(Trip,Normal,CondIn,ISnrNotTrip,DSnrNotTrip,mFlog0,mPHTD0,FlogDftHi,FaveC2,CondSwLo,pTimer0)
14: (PDLSnrDly1.trans2)
15:(Trip,Normal,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC2,CondSwLo,pTimer0)
16: (PDLDly.trans1)
17:(Trip,Pending,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC2,CondSwLo,pTimer0)
18: (PDLDly.trans2)
19:(Trip,DlyTrip,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC2,CondSwLo,pTimer0)
20: (FaveC.trans4)
21:(Trip,DlyTrip,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC1,CondSwLo,pTimer0)
22: (PDLDly.trans3)
23:(Trip,Waiting,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC1,CondSwLo,pTimer0)
24: (PDLDly.trans6)
25:(Trip,Normal,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC1,CondSwLo,pTimer0)
26: (PDLTrip.trans4)
27:(NotTrip,Normal,CondIn,ISnrNotTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC1,CondSwLo,pTimer0)
28: (PDLSnrI1.trans1)
29:(NotTrip,Normal,CondIn,ISnrTrip,DSnrTrip,mFlog0,mPHTD0,FlogDftHi,FaveC1,CondSwLo,pTimer0)
```

Fig. A1. Simulation trace of the UPPAAL output.

Line 15–19. The delayed trip occurs accordingly to the model shown in Fig. A6.

Line 19–21. The system reads the value of FaveC again (line 20). This value is in a state that can make the delayed trip status normal.

Line 21–25. Passing by the waiting state during the time interval [0.9, 1.1], the delayed trip comes to the normal state. Refer to the model in Fig. A6.

Line 25–27. The above situation causes the state of PDLTrip to be in NotTrip through the transition (PDLTrip. trans4) (line 26). Refer to Fig. A7, which shows the timed automata of the PDLTrip process.

Note that the immediate trip condition has already been activated.

Line 29. As a result, even though the immediate trip condition is activated, the process of PDLTrip stays at

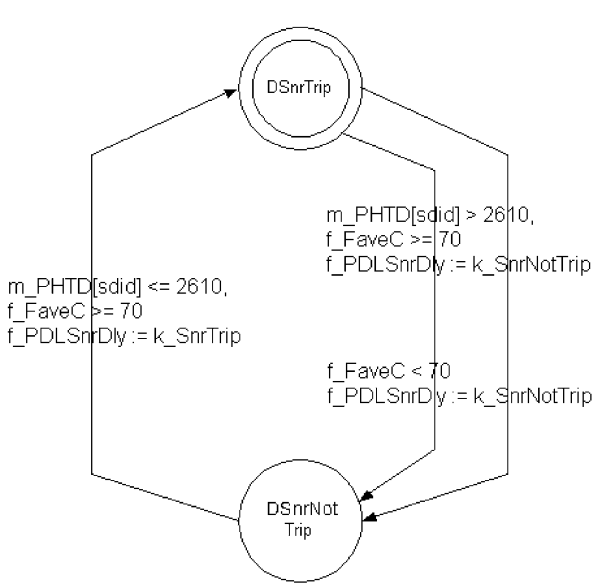

Fig. A2. Timed automata for PDLSnrDly process.

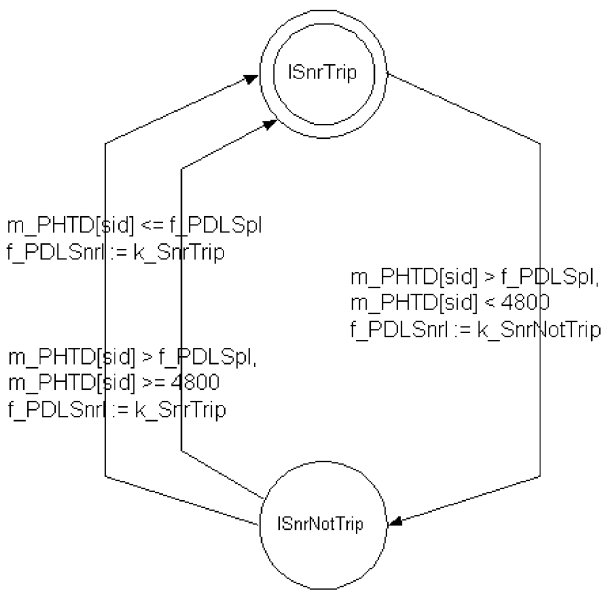

Fig. A3. Timed automata for PDLSnrI process.

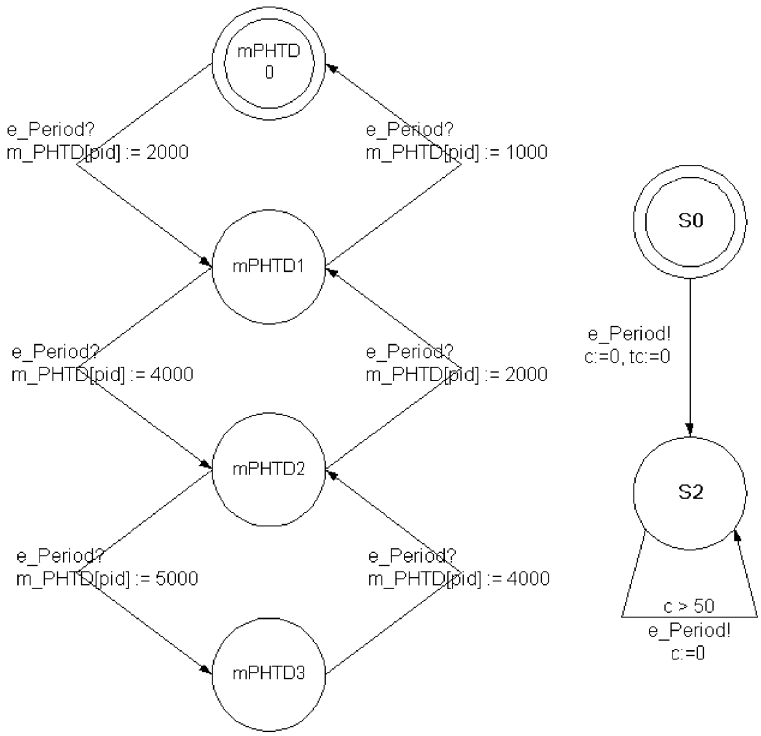

Fig. A4. Timed automata for the process of reading mPHTD and pTimer.

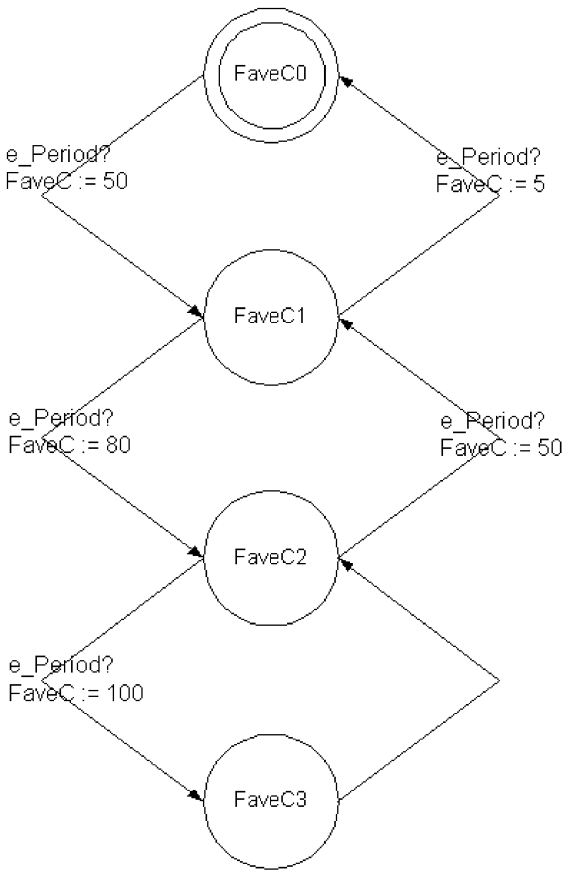

Fig. A5. Timed automata for the process of reading FaveC.

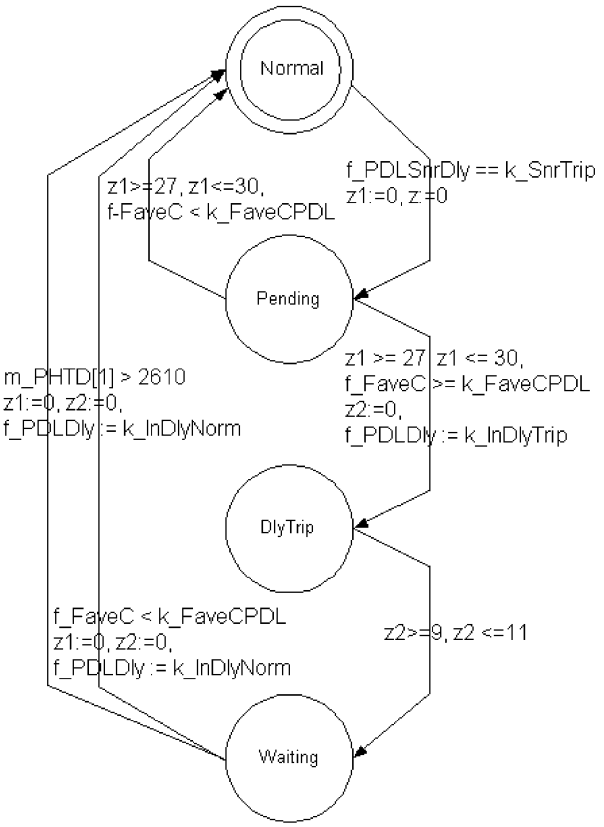

Fig. A6. Timed automata for PDLDly process.

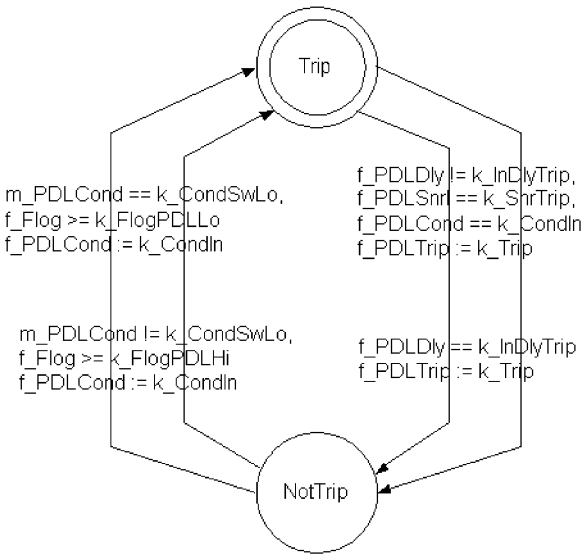

Fig. A7. Timed automata for the process of PDLTrip.

the state of NotTrip. This means that the system is in an unsafe state that may be a critical hazard.

The NotTrip state of PDLTrip means that both the immediate trip and the delayed trip are in 'Normal' state. The above counter example, however, shows a situation that the system may change its state from Trip to NotTrip when only one kind of trips changes to normal state. This is because both of the two types of trip determine the state of PDLTrip with the same priority.

References

[1] Vesely WE. Fault tree handbook. Technical report NUREG-0492, US Nuclear Regulatory Commission; 1981.

[2] Leveson NG. Safeware: system safety and computers. New York: Addison-Wesley; 1995.

[3] Kocza G, Bossche A. Automatic fault-tree synthesis and real-time trimming, based on computer models. Proc Ann Reliab Maintainability Symp 1997;71–5.

[4] WWW formal technical review (FTR) Archive, http://www.ics.hawaii.edu/~johnson/FTR/

[5] Clarke Jr.EM, Grumberg O, Peled DA. Model checking. Cambridge, MA: MIT Press; 1999.

[6] McMillan KL. Symbolic model checking: an approach to the state explosion problem. Dordrecht: Kluwer; 1993.

[7] Holzman GJ. The model checker SPIN. IEEE Trans Software Engng 1997;23(5).

[8] Bengtsson J, Larsen KG, Larsson F, Pettersson P, Yi W. UPPAAL: a tool suite for automatic verification of real-time systems. Proceedings of the Fourth DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, NJ; October 1995.

[9] Pnueli A. The temporal logic of programs. Proceedings of the 18th IEEE Symposium on Foundations of Computer Science; 1977. p. 46–77.

[10] Program functional specification, SDS2 programmable digital comparators, Wolsong NPP 2,3,4. Technical Report 86-68300-PFS-000 Rev. 2, AECL CANDU; May 1993.

[11] Software requirement specification, SDS2 programmable digital comparators, Wolsong NPP 2,3,4. Technical report 86-68350-SRS-001 Rev. 0, AECL CANDU; June 1993.

[12] Software design description, SDS2 programmable digital comparators, Wolsong NPP 2,3,4. Technical report 86-68350-SDD-001 Rev. 0, AECL CANDU; December 1993.

[13] Moss TJ. Quantitative techniques for nuclear plant safety assessment and design. Meeting on Nuclear Power Reactor Safety, Brussels, Belgium; 16 Oct 1978.s

[14] Andrews JD. The use of not logic in fault tree analysis. Qty Reliab Engng Int 2001;17(3):143–50.

[15] Alur R, Dill D. Automata for modeling real-time systems. Theor Comput Sci 1994;126(2):183–236.

[16] Ben-Ari M, Manna Z, Pnueli A. The temporal logic of branching time. Acta Inform 1983;20:207–26.