A Formal Embedded Software Verification using Software Fault Tree Analysis

Junbeom Yoo

Division of Computer Science and Engineering
Konkuk University, 1 Hwayang-dong, Gwangjin-gu, Seoul, Republic of Korea
E-mail: jbyoo@konkuk.ac.kr

Extended Abstract

In developing embedded software such as nuclear reactor protection systems (RPS), safety analysis [1] is the process performed in order to guarantee software safety, as well as development and verification processes. Fault tree analysis (FTA) [2] is one of the most widely used safety analysis techniques, often generated and applied manually. Increasing use of formal specification has made it possible to generate software fault trees mechanically, but they all have an intrinsic limitation to the information they can contain. They do not have any information beyond that captured in their specifications or source code. In this extended abstract, we used the generated software fault tree from a different standpoint, verification purpose.

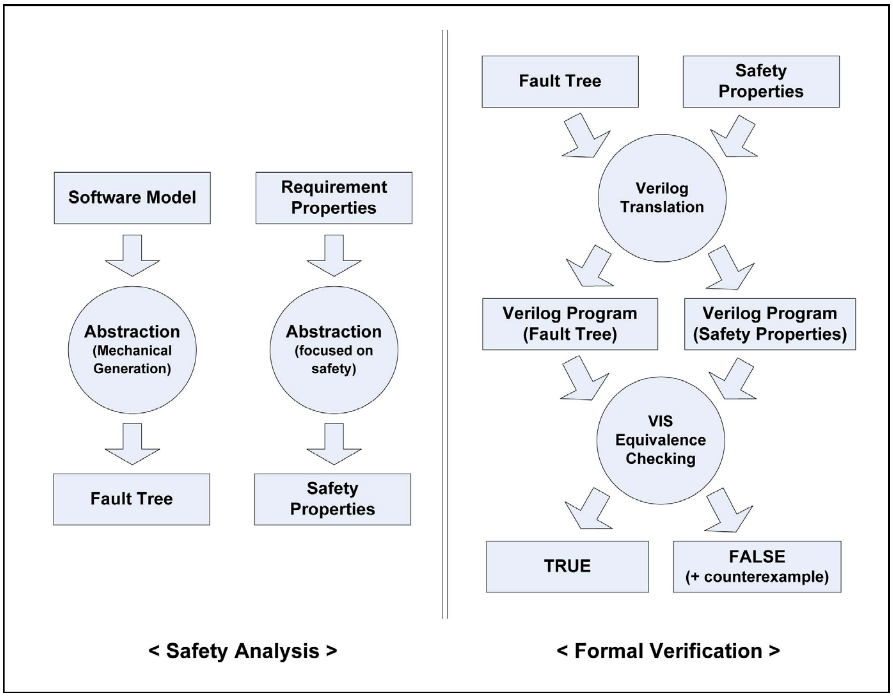


Figure 1 An overview of embedded software verification using software fault trees

Figure 1 depicts an overview of the proposed software verification technique using software fault tree as a starting point of formal verification. We regard the fault tree as an abstract model of the software, containing information only about its root-node (top failure). We therefore can check it against requirements properties quickly and analyze the verification results easily, in comparison with other verification techniques such as model checking [3]. The proposed technique translates the abstract model and properties both into Verilog programs, and performs a formal verification; VIS's combinational equivalence checking [4]. We used a prototype version of the KNICS RPS [5] in Korean nuclear power plants to demonstrate its effectiveness, and it showed that the mechanically generated fault tree is a good starting point for verifying a software model quickly against requirements properties regarding safety. We are currently focusing on developing a CASE tool, which mechanically generates software fault tree from NuSCR [6] formal requirements specification.

Acknowledgement

This research was supported 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)

References

[1] N. G. Leveson, SAFEWARE, System safety and Computers, Addison Wesley, 1995.

[2] N. G. Leveson and P. R. Harvey, “Software fault tree analysis”, Journal of Systems and Software, Vol. 3, pp. 173–181, 1983.

[3] E.M. Clarke, E.A. Emerson and A.P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications”, CM Trans. Programming Languages and Systems, Vol.8, No.2, pp.244–263, 1986.

[4] 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 and T. Villa, “VIS : A system for verification and synthesis”, In the Eighth International Conference on Computer Aided Verification, CAV ’96, pp.428–432, 1996.

[5] KAERI(Korea Atomic Energy Rearch Institute), SRS for Reactor Protection System, KNICS-RPS-SRS101 Rev.00 , 2003.

[6] J. Yoo, E. Jee and S. S. Cha, “Formal Modeling and Verification of Safety-Critical Software”, IEEE Software, Vol.26, No.3, pp.42-49, 2009.

The 5th International Symposium on Embedded Technology (ISET)
13-14 May, 2010
The Grand Hotel Daegu, Daegu, Republic of Korea

A Formal Embedded Software Verification using Software Fault Tree Analysis

JUNBEOM YOO

Dependable Software Laboratory
http://dslab.konkuk.ac.kr
KONKUK University

Contents

❖ **Software Fault Tree Analysis**

❖ **Formal Verification using Software Fault Tree**
- SFT-to-Verilog Translation
- Property-to-Verilog Translation
- VIS Equivalence Checking
  - Case Study: KNICS RPS BP (Ver.00)

❖ **Conclusion**

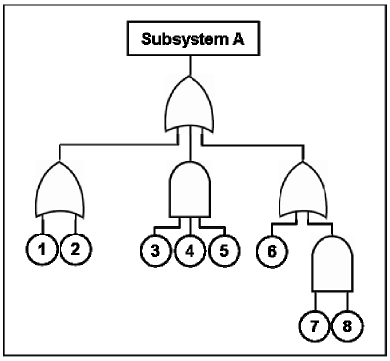
❖ **References**

2

Software Fault Tree Analysis

ReVeT

- **Fault Tree Analysis** (FTA)
  - One of widely used safety analysis techniques
  - Manually construct a fault tree and analyze with
  - Quality of FTA
    - Totally depends on experience and knowledge of FTA experts



3

Software Fault Tree Analysis

ReVeT

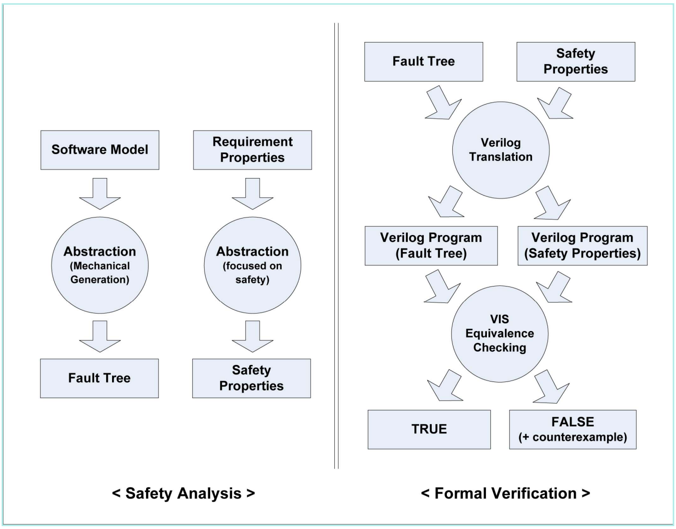
- **Software Fault Tree Analysis** (SFTA)
  - Target : Software
  - Software has no wear-out failure as hardware, since reflects developer's logic.
  - Mechanical construction of SFT is recommended.
    - from specification
      - HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [1]
      - RIDL (Reliability Imbedded Design Language) [2]
      - **NuSCR [3] (← Our concerns)**
      - Statecharts [4,5]
      - RSML [6]
    - from source code
      - Ada83/95 [7,8]
      - FBD [9]
  - Intrinsic limitation to the information they contain within

4

Formal Verification using Software Fault Tree

ReVeT

❖ **We use mechanically constructed SFTs for verification purpose.**



5

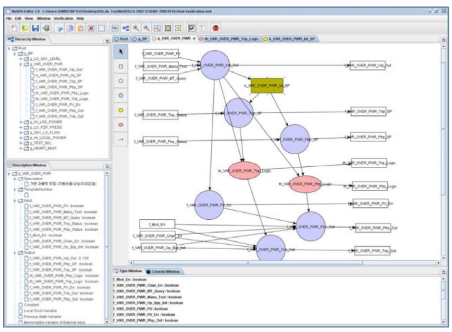
Software Fault Tree Analysis on NuSCR

ReVeT

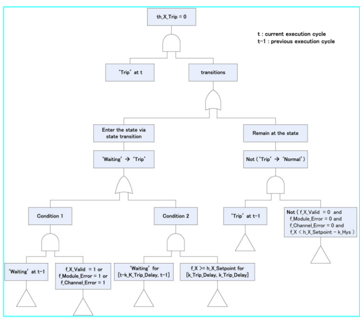
❖ **NuSCR [10]**

- Formal requirements specification language for RPS (Reactor Protection System)
- Targeting PLC (Programmable Logic Controller) software

❖ **Automatic SFT Construction from NuSCR [3]**



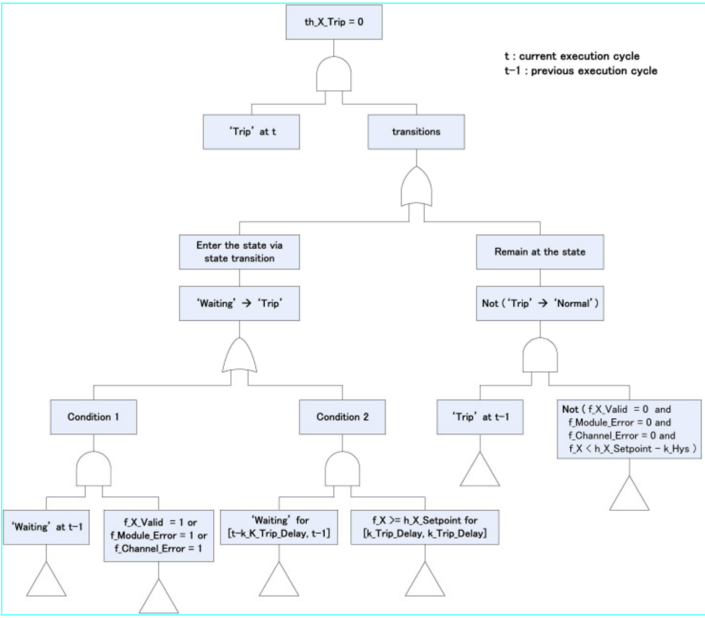
NuSCR formal specification



Software fault tree

6

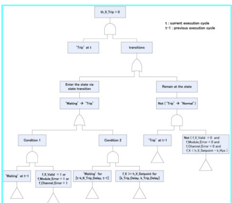
Software Fault Tree (from NuFTA [11])



7

SFTA-to-Verilog Translation

❖ **Translation from SFTA into Verilog program**



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


endmodule
```

8

Requirements Properties to Verify

ReVeT

Fairness:

1. If the value of input variables is out of bounds, the system should fire a shutdown signal immediately. (th_X_Trip := 0)
2. Only after all conditions resulting in the 'out of bounds error' are canceled, it can stop fire the shutdown signal.
3. If the trip condition regarding the operation variable f X is satisfied, it should fire a shutdown signal immediately.
4. If the trip condition regarding the operation variable f X is canceled, it should stop fire the shutdown signal immediately.

Correctness:

1. When it comes into the state 'Normal ', no shutdown signal should be fired.
2. When it comes into the state 'Waiting', no shutdown signal should be fired.
3. When it comes into or remains at the state 'Trip', a shutdown signal should be fired.

9

Property-to-Verilog Translation

ReVeT

❖ **Translation from Properties into Verilog program**

```
//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:
   ((f_X_Valid == 0 && f_Module_Error == 0 && f_Channel_Error == 0) && Current_State == 2)?1:
   ((f_X >= h_X_Setpoint) && Current_State != 2)?0:
   ((f_X < h_X_Setpoint - `k_Hys) && Current_State == 2)?1:

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

endmodule
```

VIS Equivalence Checking

ReVeT

```
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
$
```

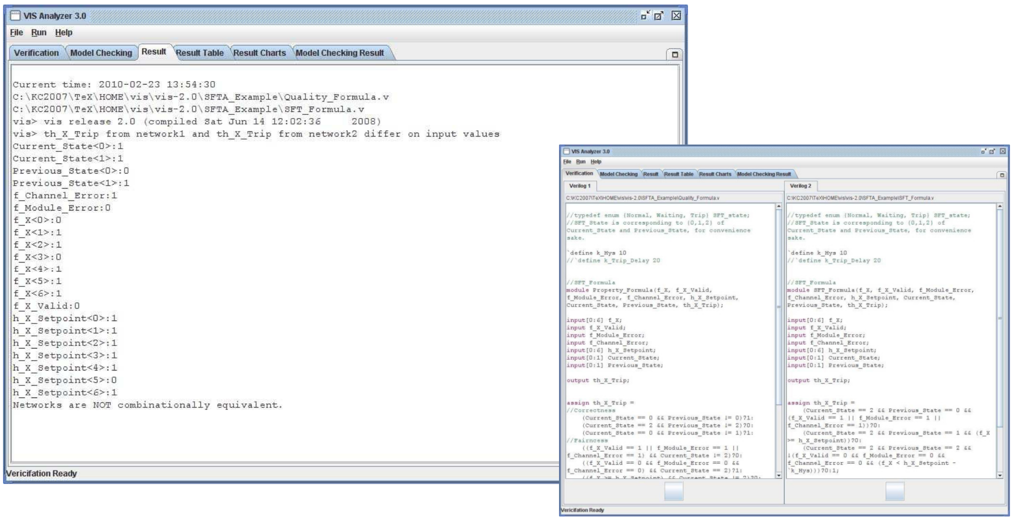
- **VIS 2.0 [12]**
  - Formal verification tool-set
    - Equivalence checking
    - Simulation
    - Model checking
    - Model synthesis
  - Verilog front-end
- **Features**
  - Powerful, but
  - No GUI

11

VIS Analyzer 3.0 [13]

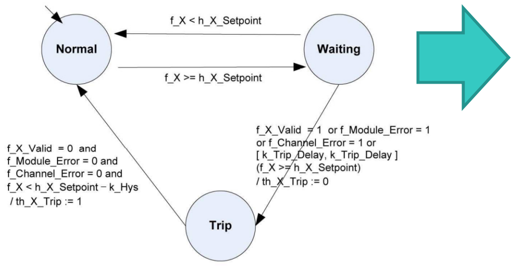
ReVeT

- **VIS assisting tool**
  - Automating verification processes
  - Visualizing analysis results

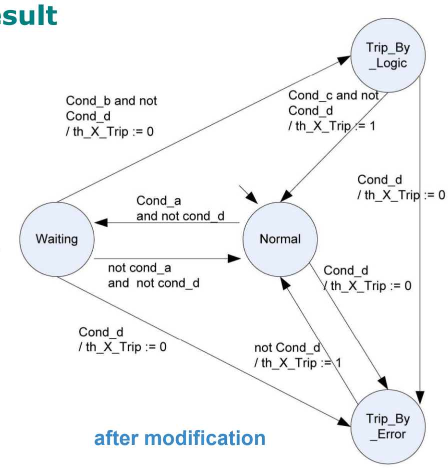


Analysis of the Verification Result

- **According to the verification result**
  - We could find the fault quickly, which had been found through full-scale verification i.e. model checking.



before (with an error)



after modification

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

13

Conclusion

- **We propose a safety-focused verification technique using software fault trees which mechanically generated from formal specifications.**
  - Software fault tree is an abstract model of its software specification, containing information only regarding the root-node of the fault tree.
  - We use a SFT as a starting point of formal verification.
- **The verification technique consists in**
  - SFT-to-Verilog translation
  - Property-to-Verilog translation
  - VIS equivalence checking
- **Case study**
  - A prototype version of KNICS RPS BP introduced in [14]

14

References

[1] 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.

[2] K. Vemuri, J. Dugan, K. Sullivan, Automatic synthesis of fault trees for computer-based systems, IEEE Transactions on Reliability 48 (4) (1999) 394–402.

[3] 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 (in Korean) 32 (12) (2005) 1178–1191.

[4] J. Yoo, S. Cha, H. S. Son, Automatic Generation of Goal-Tree from Statecharts Requirements Specification, America Nuclear Society Transactions 88 (2003) 37–38.

[5] 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.

[6] V. Ratan, K. Partridge, J. Reese, N. Leveson, Safety analysis tools for requirements specifications, in: the 7th COMPASS Workshop, 1996, pp. 149–160.

[7] N. G. Leveson, S. Cha, T. Shimeall, Safety verification of ada programs using software fault trees, IEEE Software 8 (4) (1991) 48–59.

[8] 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) - LNCS 1698/1999, 1999, pp. 226–238.

[9] 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.

[10] 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.

[11] S. Yun, D.-A. Lee, J. Yoo, NuFTA: A CASE Tool for Automatic Software Fault Tree Analysis, Transactions of the Korean Nuclear Society Spring Meeting, 2010, submitted.

[12] 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: the Eighth International Conference on Computer Aided Verification, CAV '96, 1996, pp. 428–432.

[13] 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, to be presented.

[14] J. Yoo, E. Jee, S. S. Cha, Formal Modeling and Verification of Safety-Critical Software, IEEE Software 26 (3) (2009) 42–4

15

Thank you.

Requirements and Validation
Engineering Technology Center