NuEditor – A Tool Suite for Specification and Verification of NuSCR

Jaemyung Cho, Junbeom Yoo, Sungdeok Cha

Department of Electrical Engineering and Computer Science and AITrc/SPIC/IIRTRC
Korea Advanced Institute of Science and Technology (KAIST)
373-1, Kusong-dong, Yusong-gu, Taejeon, Korea
E-mail: {jmcho, jbyoo, cha}@salmosa.kaist.ac.kr

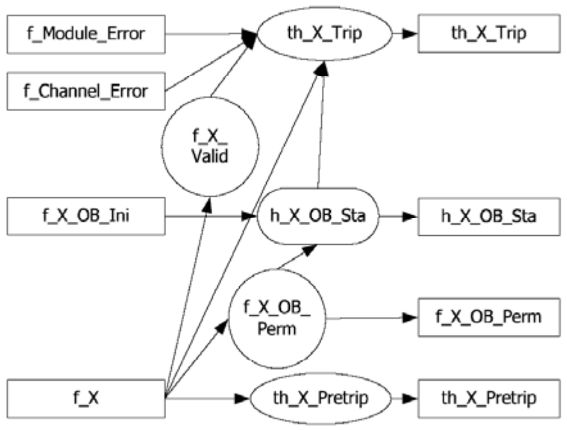
Abstract

NuEditor is a tool suite supporting specification and verification of software requirements written in NuSCR. NuSCR extends SCR (Software Cost Reduction) notation that has been used in specifying requirements for embedded safety-critical systems such as a shutdown system for nuclear power plant. SCR almost exclusively depended on fine-grained tabular notations to represent not only computation-intensive functions but also time- or state-dependent operations. As a consequence, requirements became excessively complex and difficult to understand. NuSCR supports intuitive and concise notations. For example, automata is used to capture time or state-dependent operations, and concise tabular notations are made possible by allowing complex but proven-correct equations be used without having to decompose them into a sequence of primitive operations. NuEditor provides graphical editing environment and supports static analysis to detect errors such as missing or conflicting requirements. To provide high-assurance safety analysis, NuEditor can automatically translate NuSCR specification into SMV input so that satisfaction of certain properties can be automatically determined based on exhaustive examination of all possible behavior. NuEditor has been programmed to generate requirements as an XML document so that other verification tools such as PVS can also be used if needed. We have used NuEditor to specify a trip logic of RPS(Reactor Protection System) BP(Bistable Processor) and verify its correctness. It is a part of software-implemented nuclear power plant shutdown system. Domain experts found NuSCR and NuEditor to be useful and qualified for industrial use in nuclear engineering.

1. Introduction

Many validation and verification techniques (e.g. inspection, fault tree analysis, simulation, model checking, etc) have been proposed to ensure safety. In nuclear power plant control systems, software safety became a critical issue as traditional RLL(Relay Ladder Logic)-based analog systems are replaced by digital controllers [2]. KNICS project [3] in Korea is developing DPPS(Digital Plant Protection System) RPS(Reactor Protection System) which is classified as being safety-critical by government regulation authority. To maximize safety of RPS software, proven-effective formal methods are being used. For example, SCR-style notation was previously used to specify software requirements for Wolnsung SDS2, a shutdown system currently in service at a different plant in Korea. Experts who performed critical analysis on SCR and other formal specification languages came to the conclusion that SCR-like notation is well-suited for specifying and verifying requirements for RPS but that the notation in its current form is too verbose to be effectively used. Furthermore, availability of SCR* toolset was unsatisfactory from the viewpoint of KNICS project management office. Therefore, an effort was initiated to (1) customize SCR so that characteristics unique to nuclear engineering domain are best reflected in the design of a specification language; and (2) develop a tool suite, NuEditor, to integrate graphical editing capability and formal verification environment. In addition to performing built-in completeness and consistency analysis on NuSCR specification, NuEditor can generate SMV [11] input program automatically so that one can perform model checking with minimal intervention. It also generates XML output that is used as input to PVS for deductive verification of structural and functional properties [12].

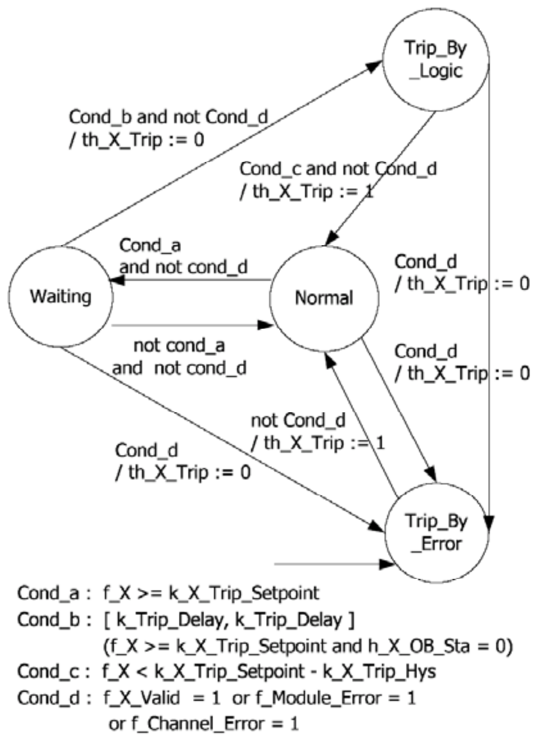
To find out if NuSCR and NuEditor are useful enough to nuclear engineers, we conducted a joint study with a group of domain experts in which trip logic of RPS(Reactor Protection System) BP(Bistable Processor) was specified and verified. This paper introduces key features of NuEditor and reports our experience from the case study. Section 2 briefly introduces NuSCR, and section 3 provides an overview of NuEditor features. After reporting our experience with NuEditor from the case study in section 4, we conclude the paper and discuss planned extensions to NuEditor.



(a) FOD for *g_Fixed_Setpoint_Rising_Trip_with_OB*

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

(b) SDT for function variable node *f_X_Valid*


Trip_By_Logic
Cond_b and not Cond_d / th_X_Trip := 0
Cond_c and not Cond_d / th_X_Trip := 1
Cond_a and not cond_d
Waiting
Normal
Cond_d / th_X_Trip := 0
not cond_a and not cond_d
Cond_d / th_X_Trip := 0
not Cond_d / th_X_Trip := 1
Cond_d / th_X_Trip := 0
Trip_By_Error
Cond_a : f_X >= k_X_Trip_Setpoint
Cond_b : [k_Trip_Delay, k_Trip_Delay] (f_X >= k_X_Trip_Setpoint and h_X_OB_Sta = 0)
Cond_c : f_X < k_X_Trip_Setpoint - k_X_Trip_Hys
Cond_d : f_X_Valid = 1 or f_Module_Error = 1 or f_Channel_Error = 1

(c) TTS for timed history variable node *th_X_Trip*

Figure 1. NuSCR Specifications Example

2. NuSCR

NuSCR [4], as noted earlier, customizes SCR (Software Cost Reduction) [5] to nuclear engineering industry. NuSCR, based on SCR-style AECL notation [6] used in specifying requirements for Wolsung SDS2, uses FOD(Function Overview Diagram) to capture high-level data flows. In addition, three basic constructs - function variable, history variable, and timed history variable - are defined by SDT(Structured Decision Table), FSM(Finite State Machine), and TTS(Timed Transition System), respectively [7]. NuSCR improves the readability of specification and enhances expressiveness by supporting intuitive notations. Details on formal definition of NuSCR syntax and semantics are found in [4].

Figure 1(a) is a FOD for *g_Fixed_Setpoint_Rising_Trip_with_OB*, fixed set-point rising trip logic in BP, where *g_* denotes the group prefix. Boxed nodes represent inputs and outputs. SDT, shown in Figure 1(b), defines function variable *f_X_Valid* appearing in the FOD. If the value of *f_X* is between *k_X_MIN* and *k_X_MAX*, the output value *f_X Valid* is 0, indicating normal case. Otherwise output value is 1. NuSCR allows multiple and related terms be written together on the same row. That is, in the AECL-notation, one would have no option but to divide into into two rows: (*f_X >= k_X_MIN)* and (*f_X <= k_X_MAX)*. This example is too trivial for developer to appreciate the difference in expressiveness. However, in the Wolsung SDS2, which was considerably simpler in complexity than KNICS RPS, the most complex SDT consisted of 16 rows and 12 columns because complex equations had to be decomposed into “primitive” fragments. Domain experts repeatedly emphasized that mathematical equations used in trip logics, no matter how complex they are, are well-understood and proven-correct as a whole to domain experts and that they need not be artificially fragmented in the specification.

Figure 1(c), TTS for *th_X_Trip,* illustrates how behavior of timed-history variable node is captured. It is interpreted as follows: “If condition *f_X* ≥ *k_X_Trip_Setpoint* is satisfied in state *Normal*, it enters *Waiting* state. If the condition remains true for *k_Trip_Delay* period while in *Waiting* state, system generates the trip signal 0. If *f_X_Valid, f_Module_Error*, or *f_Channel_Error* occur, then trip signal is immediately produced. In the state *Trip_By_Error* or *Trip_By_Logic*, if the trip conditions are canceled, system returns to *Normal* state and the output 1 is generated." The TTS expression in *Cond_b [k_Trip_Delay, k_Trip_Delay]* means that the condition must remain true for *k_Trip_Delay* unit times. In AECL-style notation, behavior related to time-dependent state transition was written in tabular notation, and domain experts preferred automata notation to tabular notation.

Similarly, *h_X_OB_Sta,* shown in Figure 1(a), is a history variable node defined as FSM. FSM is same as

TTS except that time constraints are missing. All constructs in NuSCR, s.t. FOD, SDT, FSM, and TTS are familiar notations to domain engineers and software developers. NuSCR has been evaluated as being easy to specify and understand by domain engineers [8].

3. NuEditor Features and Capabilities

Main functionalities of NuEditor are shown in Figure 2. NuEditor, developed in Java, is platform independent. All constructs in NuSCR (e.g., FOD, SDT, FSM, and TTS) can be graphically edited using NuEditor. Various nodes are colored differently so that they roles are visually apparent. NuEditor stores models in hierarchically organized folders, as shown on the left side of the tool window, so that requirements for large and complex industrial systems can be conveniently organized. Users can add annotations and comments as needed. In addition to a specification editor, consistency and completeness checker was included. Figure 3 (a) shows FOD and FSM editing windows, and Figure 3 (b) shows SDT window and XML generator window. As shown in Figure 4, analysis on structural correctness is automated. That is, when a group node is expanded in a separate page, inputs and outputs declared at a higher-level node are shown. If detailed specification of inputs and outputs on that page neglects to use them all, error message pops up to warn users that usage of variables is inconsistent. Variables can also be dragged so that users need not explicitly type variable names repeatedly.

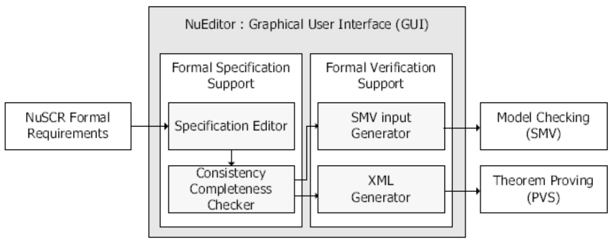
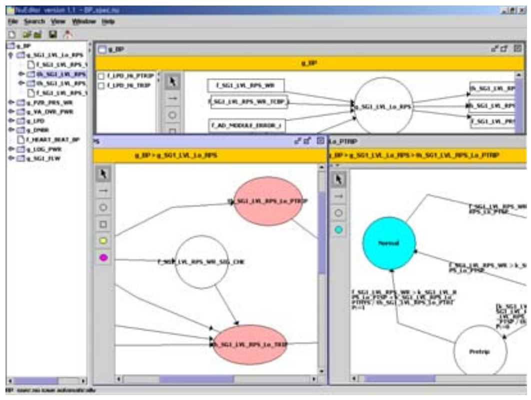

NuEditor : Graphical User Interface (GUI)
Formal Specification Support
Formal Verification Support
NuSCR Formal Requirements
Specification Editor
SMV input Generator
Model Checking (SMV)
Consistency Completeness Checker
XML Generator
Theorem Proving (PVS)

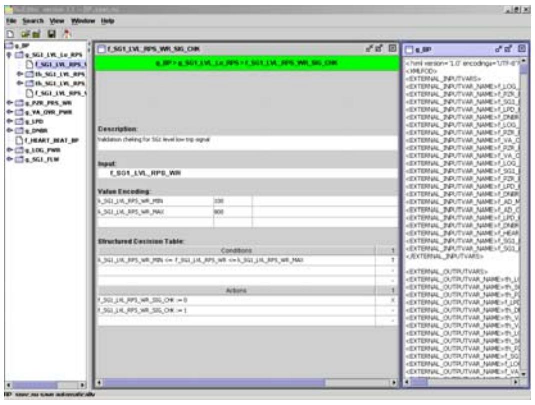
Figure 2. NuEditor Functionality

To support formal verification, NuEditor includes a XML(Extensible Markup Language) generator and a SMV input generator. The XML generator is used to prove the structural and functional properties of NuSCR specification using PVS [9,10]. Theorem proving[15] is a deductive verification method. While powerful, proof sessions are often lengthy and tedious in practice. Fortunately, modern theorem provers like PVS provide excellent support in proof automation and development of proof strategies. To best utilize capabilities of tools like PVS, NuEditor generates XML documents which can then be used as input to other applications. XML documents, for example, can be used in developing design specification written in FBD(Function Block Diagram) notations [13] as is the case in the KNICS project [14].

The SMV input generator is used to check if specification satisfies certain properties written in temporal logic. Model checking[16] is a technique enabling "push-button" verification based on exhaustive search of possible behavior. Model checking is becoming popular in industry because (1) it is automated; and (2) a counterexample is generated if the property does not hold in the specification. Counterexample can reveal the presence of subtle flaws in the specification or can be used to automatically construct test cases. SMV is arguable the most widely used model checker to date, and NuEditor can automatically generate input to SMV model checker [11]. User simply needs to execute SMV software (e.g., Cardence SMV), load the specification file and property file, and select verify all menus in the option.

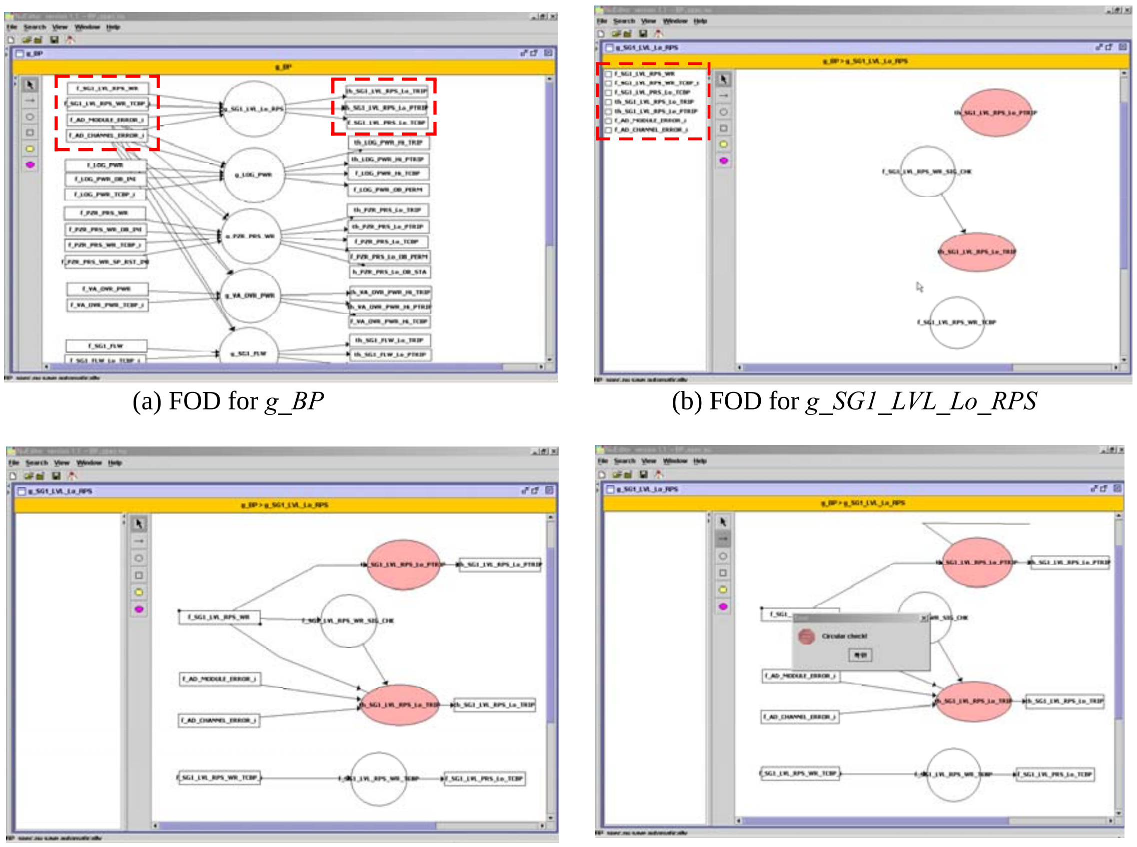


(a) FOD, FSM Editing Window



(b) SDT, XML Generation Window

Figure 3. Screen shot of NuEditor



(a) FOD for *g_BP*

(b) FOD for *g_SG1_LVL_Lo_RPS*

(c) FOD for *g_SG1_LVL_Lo_RPS*

(d) Circular dependency checking

Figure 4. Consistency and Completeness Checking

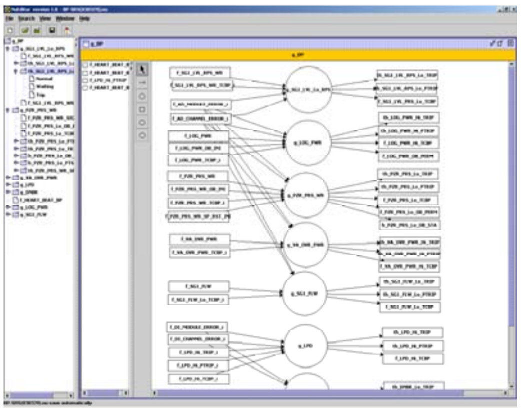
4. Case Study

KNICS RPS includes RPS(Reactor Protection System), ESF-CCS(Engineering Safety Features - Component Control System), and ATIP(Automatic test and Interface Processor) as major components. RPS is designed to protect the reactor, while ESF-CCS is intended to reduce the influence of other accidents including loss of coolant. ATIP tests RPS and ESF-CCS automatically. In this section, we present how NuEditor was used in specifying requirements for BP (Bistable Logic) logic. We performed model checking of BP specification.

RPS BP periodically accepts inputs from 18 different safety sensors installed in the system and performs necessary comparison against predefined trip logics and threshold values. For example, Figure 5 is a part of NuSCR specification for RPS BP. In figure 5 (a), *g_BP*, a group node, is decomposed in Figure 5(b). NuSCR software requirements specification for KNICS BP is about 400 pages, and it took 5 months by a number of domain experts.

We present the results of model checking fixed set-point rising trip logic with operating bypass. The logic description written in natural language took about four pages. Translation rules used in NuEditor are similar to those proposed in [18, 19, 20, 21]. [18] translates SCR specification into SMV input language, whereas [19] translates SCR specification into language accepted by SPIN [20] Since NuSCR, due to inclusion of FSM and TTS in its notation, is more analogous to RSML than SCR, our rule were mainly based on translation rules for RSML [21]. More detailed translation methods are described in [17].

Figure 6 shows SMV input program for *th_X_Trip* shown earlier in Figure 1 (c). Since variables in SMV must have finite discrete values, user must abstract infinite values (e.g. *f_X* at line 8) as integer although *f_X* actually returns a real number as its result. Constants defined in the systems (lines 42 through 44) are separately managed by NuEditor. Lines 35 through 39 and 51 reflect TTS specification including timer variables, i.e. *time_1* is a clock variable in TTS and line 51 is an action triggered by the variable.


(a) FOD for *g_BP*

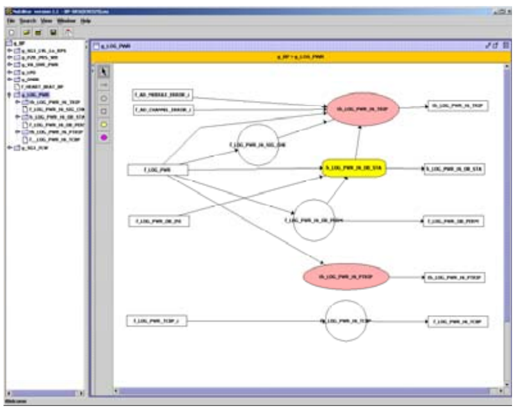
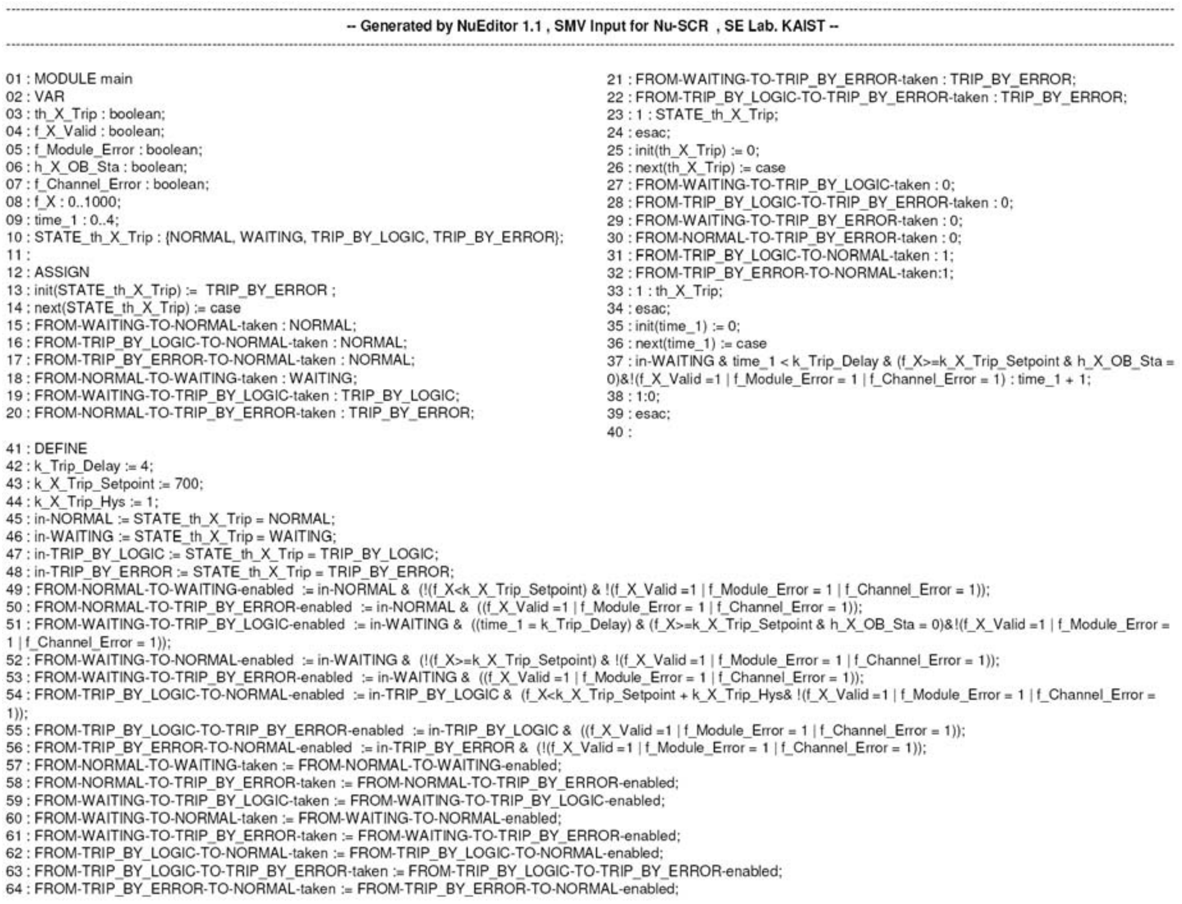

(b) FOD for *g_LOG_PWR*

Figure 5. FOD for RPS *g_BP*



```
-- Generated by NuEditor 1.1 , SMV Input for Nu-SCR  , SE Lab. KAIST --

MODULE main
VAR
th_X_Trip : boolean;
f_X_Valid : boolean;
f_Module_Error : boolean;
h_X_OB_Sta : boolean;
f_Channel_Error : boolean;
f_X : 0..1000;
time_1 : 0..4;
STATE_th_X_Trip : {NORMAL, WAITING, TRIP_BY_LOGIC, TRIP_BY_ERROR};

ASSIGN
init(STATE_th_X_Trip) :=  TRIP_BY_ERROR ;
next(STATE_th_X_Trip) := case
FROM-WAITING-TO-NORMAL-taken : NORMAL;
FROM-TRIP_BY_LOGIC-TO-NORMAL-taken : NORMAL;
FROM-TRIP_BY_ERROR-TO-NORMAL-taken : NORMAL;
FROM-NORMAL-TO-WAITING-taken : WAITING;
FROM-WAITING-TO-TRIP_BY_LOGIC-taken : TRIP_BY_LOGIC;
FROM-NORMAL-TO-TRIP_BY_ERROR-taken : TRIP_BY_ERROR;
FROM-WAITING-TO-TRIP_BY_ERROR-taken : TRIP_BY_ERROR;
FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-taken : TRIP_BY_ERROR;
1 : STATE_th_X_Trip;
esac;
init(th_X_Trip) := 0;
next(th_X_Trip) := case
FROM-WAITING-TO-TRIP_BY_LOGIC-taken : 0;
FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-taken : 0;
FROM-WAITING-TO-TRIP_BY_ERROR-taken : 0;
FROM-NORMAL-TO-TRIP_BY_ERROR-taken : 0;
FROM-TRIP_BY_LOGIC-TO-NORMAL-taken : 1;
FROM-TRIP_BY_ERROR-TO-NORMAL-taken:1;
1 : th_X_Trip;
esac;
init(time_1) := 0;
next(time_1) := case
in-WAITING & time_1 < k_Trip_Delay & (f_X>=k_X_Trip_Setpoint & h_X_OB_Sta = 0)&!(f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1) : time_1 + 1;
1:0;
esac;

DEFINE
k_Trip_Delay := 4;
k_X_Trip_Setpoint := 700;
k_X_Trip_Hys := 1;
in-NORMAL := STATE_th_X_Trip = NORMAL;
in-WAITING := STATE_th_X_Trip = WAITING;
in-TRIP_BY_LOGIC := STATE_th_X_Trip = TRIP_BY_LOGIC;
in-TRIP_BY_ERROR := STATE_th_X_Trip = TRIP_BY_ERROR;
FROM-NORMAL-TO-WAITING-enabled  := in-NORMAL &  (!(f_X<k_X_Trip_Setpoint) & !(f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-NORMAL-TO-TRIP_BY_ERROR-enabled  := in-NORMAL &  ((f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-WAITING-TO-TRIP_BY_LOGIC-enabled  := in-WAITING &  ((time_1 = k_Trip_Delay) & (f_X>=k_X_Trip_Setpoint & h_X_OB_Sta = 0)&!(f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-WAITING-TO-NORMAL-enabled  := in-WAITING &  (!(f_X>=k_X_Trip_Setpoint) & !(f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-WAITING-TO-TRIP_BY_ERROR-enabled  := in-WAITING &  ((f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-TRIP_BY_LOGIC-TO-NORMAL-enabled  := in-TRIP_BY_LOGIC &  (f_X<k_X_Trip_Setpoint + k_X_Trip_Hys& !(f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-enabled  := in-TRIP_BY_LOGIC &  ((f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-TRIP_BY_ERROR-TO-NORMAL-enabled  := in-TRIP_BY_ERROR &  (!(f_X_Valid =1 | f_Module_Error = 1 | f_Channel_Error = 1));
FROM-NORMAL-TO-WAITING-taken := FROM-NORMAL-TO-WAITING-enabled;
FROM-NORMAL-TO-TRIP_BY_ERROR-taken := FROM-NORMAL-TO-TRIP_BY_ERROR-enabled;
FROM-WAITING-TO-TRIP_BY_LOGIC-taken := FROM-WAITING-TO-TRIP_BY_LOGIC-enabled;
FROM-WAITING-TO-NORMAL-taken := FROM-WAITING-TO-NORMAL-enabled;
FROM-WAITING-TO-TRIP_BY_ERROR-taken := FROM-WAITING-TO-TRIP_BY_ERROR-enabled;
FROM-TRIP_BY_LOGIC-TO-NORMAL-taken := FROM-TRIP_BY_LOGIC-TO-NORMAL-enabled;
FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-taken := FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-enabled;
FROM-TRIP_BY_ERROR-TO-NORMAL-taken := FROM-TRIP_BY_ERROR-TO-NORMAL-enabled;
```

Figure 6. Generated SMV input program *for th_X_Trip* in Figure 1 (c)

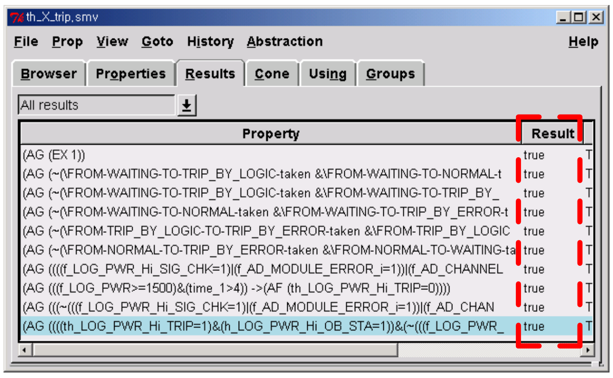
The following properties were verified using SMV:

① System is free from deadlock.
② Conflicting transitions are never enabled simultaneously.
③ If module error, channel error, or input value error occur, trip signal is generated immediately.
④ Trip signal is generated if the processing value rises above the predefined set-point, and the condition lasts for some predefined time.
⑤ If trip conditions aren't satisfied, then trip signal shall never be fired.
⑥ Trip signal is never fired during operating bypass.

Properties, written in CTL formula, are as follows. It must be noted that there are no automated support built in NuEditor in specifying properties. Users are expected to be familiar with basics of temporal logic and its operators.

① Deadlock-freeness
SPEC AG EX 1
② Non-determinism
SPEC AG! (FROM-WAITING-TO-TRIP_BY_LOGIC-taken & FROM-WAITING-TO-NORMAL-taken)
SPEC AG! (FROM-WAITING-TO-TRIP_BY_LOGIC-taken & FROM-WAITING-TO-TRIP_BY_ERROR-taken)
SPEC AG! (FROM-WAITING-TO-NORMAL-taken & FROM-WAITING-TO-TRIP_BY_ERROR-taken)
SPEC AG! (FROM-WAITING-TO-NORMAL-taken & FROM-WAITING-TO-TRIP_BY_ERROR-taken)
SPEC AG! (FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-taken & FROM-TRIP_BY_LOGIC-TO-NORMAL-taken)
SPEC AG! (FROM-NORMAL-TO-TRIP_BY_ERROR-taken & FROM-NORMAL-TO-WAITING-taken)
③ Trip occurred by error
SPEC AG ((f_Channel_Error = 1 | f_Module_Error = 1) → AF th_X_Trip = 0)
④ Trip occurred by logic
SPEC AG(((f_X > k_X_Trip_Setpoint) & (time_1 > 4)) → AF th_X_Trip = 0)
⑤ Normal status
SPEC AG((!(f_Channel_Error = 1 | f_Module_Error = 1 | f_X_Valid = 1) & (f_X <= k_X_Trip_Setpoint)) → AF th_X_trip = 1)
⑥ Trip in operating bypass
SPEC AG((h_X_OB_Sta = 1 & ! (f_Channel_Error = 1 | f_Module_Error = 1 | f_X_Valid = 1) & AF AX th_X_Trip = 1) → AF AX th_X_Trip = 1)

Figure 7 shows how SMV-based model checking results look like. Results marked TRUE indicate that the property is satisfied in all possible system spaces. In this case study, all properties are proved to be true using SMV model checker, so we can confirm that RPS model satisfies properties (1) through (6).


th_X_trip.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
All results

Property	Result	
(AG (EX 1))	true	T
(AG (~(\FROM-WAITING-TO-TRIP_BY_LOGIC-taken &\FROM-WAITING-TO-NORMAL-t	true	T
(AG (~(\FROM-WAITING-TO-TRIP_BY_LOGIC-taken &\FROM-WAITING-TO-TRIP_BY_	true	T
(AG (~(\FROM-WAITING-TO-NORMAL-taken &\FROM-WAITING-TO-TRIP_BY_ERROR-t	true	T
(AG (~(\FROM-TRIP_BY_LOGIC-TO-TRIP_BY_ERROR-taken &\FROM-TRIP_BY_LOGIC	true	T
(AG (~(\FROM-NORMAL-TO-TRIP_BY_ERROR-taken &\FROM-NORMAL-TO-WAITING-ta	true	T
(AG ((((f_LOG_PWR_Hi_SIG_CHK=1)|(f_AD_MODULE_ERROR_i=1))|(f_AD_CHANNEL	true	T
(AG (((f_LOG_PWR>=1500)&(time_1>4)) ->(AF (th_LOG_PWR_Hi_TRIP=0))))	true	T
(AG (((~(((f_LOG_PWR_Hi_SIG_CHK=1)|(f_AD_MODULE_ERROR_i=1))|(f_AD_CHAN	true	T
(AG ((((th_LOG_PWR_Hi_TRIP=1)&(h_LOG_PWR_Hi_OB_STA=1))&(~(((f_LOG_PWR_	true	T

Figure 7. Verification result of *th_X_Trip*

5. Conclusions and Future Work

In this paper, we presented key features of NuEditor, an integrated tool suite to perform both specification and verification of requirements specification written in NuSCR. The NuEditor includes a graphical editor, consistency and completeness checker, XML output generator, and SMV input generator. NuEditor provides graphical and user-friendly interface and relieves engineers from tedious and uninteresting work. It allows them to work on more creative tasks. Automated consistency checks save considerable time of developers and reviewers. It also increases confidence that specification is correct by allowing engineers to enjoy benefit of formal methods. NuEditor tool was well liked by nuclear engineers, and addition of simulation and backward analysis capabilities would further improve its usefulness in real applications like KNICS.

Acknowledgement

This research was partially supported by the Advanced Information Technology Research Center(AITrc), Software Process Improvement Center(SPIC), and the Internet Intrusion Response Technology Research Center(IIRTRC).

References

[1] N. G. Leveson, “SAFEWARE, System Safety and Computer,” Addison Wesley, 1995.
[2] “Digital Instrumentation and Control Systems in Nuclear Power Plants: safety and reliability issues”, U.S. NRC, National Academy Press, 1997.
[3] KNICS(Korea Nuclear Instrumentation and Control System Research and Development Center), Available: http:// www.knics.re.kr
[4] J. Yoo, T. Kim, S. Cha, J. Lee, and H. S. Son, “A Formal Software Requirements Specification Method for Digital Nuclear Plants Protection Systems,” Journal of Systems and Software, accepted.
[5] K. L. Heninger, “Specifying software requirements for complex systems: New techniques and their application,” IEEE Trans. Software Engineering, vol. SE-6, no. 1, pp. 2-13, 1980.
[6] A. J. Schouwen Van, D. Panas, and J. Madey, “Documentation of Requirements for Computer Systems,” in proc. IEEE International Symposium on Requirements Engineering, pp. 198-207, 1993.
[7] T. A. Henzinger, Z Manna, and A. Pnueli, “Timed Transition Systems,” in proc. REX Workshop, pp.226-251, 1991.
[8] J. Yoo, Y. Oh, S. Cha, and C. Kim, “Toward the Formal Software Requirements Specification for Digital Reactor Protection Systems,” IEEE trans. Nuclear Science, submitted.
[9] T. Kim and S. Cha, “Automatic Structural Analysis of SCR-style Software Requirements Specifications using PVS,” Journal of Software Testing, Verification, and Reliability, vol. 11, no. 3, pp. 143-163, 2001.
[10] T. Kim, D. Stringer-Calvert, and S. Cha, "Formal Verification of Functional Properties of an SCR-style Software Requirements Specification using PVS," Reliability Engineering and System Safety, submitted.
[11] K. L. McMillan, “Symbolic Model Checking”, Kluwer Academic Publishers, 1993.
[12] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, “A tutorial introduction to PVS,” Workshop on Industrial-

Strength Formal Specification Techniques (WIFT '95), pp. 1-112, 1995.
[13] R. W. Lewis, "Programming Industrial Control Systems Using IEC 1131-3," The Institution of Electrical Engineers, London, United Kingdom, 1995.
[14] J. Cho, Y. Oh, J. Yoo, and S. Cha, "KAIST Software Development Framework for Nuclear-Domain," 29th KISS conference, spring, 2002.
[15] D. V. Dalen, "Logic and Structure," Springer-Verlag, 3th edition, 1993.
[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent machine using temporal logic specifications," ACM Trans. Programming Language and systems, vol. 8, no. 2, pp. 244-263, 1986.
[17] J. Cho, "NuEditor : An Environment for NuSCR Specification and Verification," MS. Thesis. Korea Advanced Institute of Science and Technology(KAIST), 373-1, Kusong-dong, Yusong-gu, Taejon, Korea, Feb. 2002.
[18] J. M. Atlee and M. A. Buckley, "A logic-model semantics for SCR software requirements," In Proc International Symposium on Software Testing and Analysis, pp. 280-292, January 1996.
[19] B. Ramesh and C. L. Heitmeyer, "Model Checking Complete Requirements Specifications Using Abstraction," Automated Software Engineering, vol. 6, no. 1, pp. 37-68, January 1999.
[20] G. J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage Preserving Reduction Strategies for Reachabily Analysis," In proc. IFIP/WG6.1 Symposium, Protocol Specification, Testing, and Verification(PSTV92), pp. 349-364, 1992.
[21] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese, "Model checking large software specification," IEEE Transaction on Software Engineering, vol.24, no.7, 1998.