Journal of the Korean Nuclear Society
Volume 31, Number 2, pp.157~171, April 1999

Mathematical Verification of a Nuclear Power Plant Protection System Function with Combined CPN and PVS

Seo Ryong Koo, Han Seong Son and Poong Hyun Seong
Korea Advanced Institute of Science and Technology
373-1, Gusong-Dong, Yusong-Gu, Taejon,305-701, Korea

phseong@sorak.kaist.ac.kr

(Received June 20, 1998)

Abstract

In this work, an automatic software verification method for Nuclear Power Plant (NPP) protection system is developed. This method utilizes Colored Petri Net (CPN) for system modeling and Prototype Verification System (PVS) for mathematical verification. In order to help flow-through from modeling by CPN to mathematical proof by PVS, an information extractor from CPN models has been developed in this work. In order to convert the extracted information to the PVS specification language, a translator also has been developed. ML that is a higher-order functional language programs the information extractor and translator. This combined method has been applied to a protection system function of Wolsong NPP SDS2(Steam Generator Low Level Trip). As a result of this application, we could prove completeness and consistency of the requirement logically. Through this work, in short, an axiom or lemma based-analysis method for CPN models is newly suggested in order to complement CPN analysis methods and a guideline for the use of formal methods is proposed in order to apply them to NPP Software Verification and Validation.

Key Words : mathematical verification, CPN, PVS, formal method

1. Introduction

In safety critical systems, use of computers provides many potential advantages. These advantages include more sophisticated safety algorithms, improved availability, easier maintenance, reduced installation costs, ease of modification, and potential for reuse. However, because of the safety critical nature of the application such as NPP protection system, these advantages have to be weighed against the problems of ensuring that the computer system can be assured adequately.

The use of digital systems is also on increase in nuclear industry in recent years. Therefore, the importance of system V&V (Verification and Validation) is more emphasized in view of the nuclear safety. Many activities are required for NPP software V&V. IEEE std. 1012-1986 defines software V&V activities. This standard classifies the software V&V activities by tasks: Management of V&V, concept phase V&V, requirements phase

V&V, design phase V&V, implementation phase V&V, test phase V&V, installation and checkout phase V&V and operation and maintenance phase V&V tasks are required [9]. Of these tasks, our work concentrates on the requirements phase V&V task since the requirements phase is the most important of all the software development phases. It is well-known that the most of software faults occur at the requirements phase.

A number of different techniques have been used to verify and validate safety-related systems. Of them, formal methods have many superior properties. Formal methods are approaches, based on the use of mathematical techniques and notations, for describing and analyzing properties of software systems. Examples of formal methods are VDM, Z, CCS, CSP, I/O Automata, and Petri Nets [1]. However, there are few formal methods that support both graphical representation and mathematical proof simultaneously, though visualization and mathematical proof are essential properties of them.

In this work, an automatic software verification method for NPP protection system is developed. This method utilizes CPN for modeling and PVS for mathematical verification. CPN supports visualization and PVS does mathematical proofs. For the safety-critical protection systems, complete analysis of the system is needed since an error in the requirements may result in serious faults of software. CPN has been proved as an adequate tool for requirement analysis [2]. CPN has many advantages such as rapid prototyping and visualization of requirements. However, CPN is not proper for the mathematical verification of the system. That is, CPN has many limitations in analysis methods such as occurrence graphs, place and transition invariants, reduction rules. In order to complement these limitations, an axiom or lemma based-analysis method is suggested in this work. PVS is used for the mathematical verification through the axiom or lemma based-analysis method.

In this work, first, requirements of Steam Generator Low Level Trip (one of the Wolsung NPP SDS2 parameter) are modeled with Design/CPN. This model enables also the easy communication between users and system developers. Next, PVS specification, which is translated from CPN model, is verified mathematically. The main focus in this work is on the flow-through from CPN model to PVS specification. An information extractor and a translator have been developed for this purpose. We introduce some major aspect of CPN and PVS for this work in section 2. In section 3, the flow-through from CPN model to PVS specification with the extractor and translator is described. An automatic software verification tool is applied to Wolsung NPP SDS2 function in section 4.

2. Colored Petri Net and Prototype Verification System

2.1. CPN and Design/CPN

Petri Net is a language that has been used in modeling and analyzing the system. Petri Net has expressions of concurrency and formal semantics. In addition, Petri Net can visualize the actual system with ease. However, Petri Net is so basic that the ability of expression is limited and CPN has been developed to overcome this limit. In CPN, color refers to the types of data associated with tokens and is comparable to data types in programming languages. Design/CPN is a powerful tool for the Colored Petri Net. The version of CPN used in Design/CPN incorporates variables (representing the binding of identifiers to specific colored tokens), arc inscriptions (expressions), and the code associated with transitions. In Design/CPN, CPN is a graphical

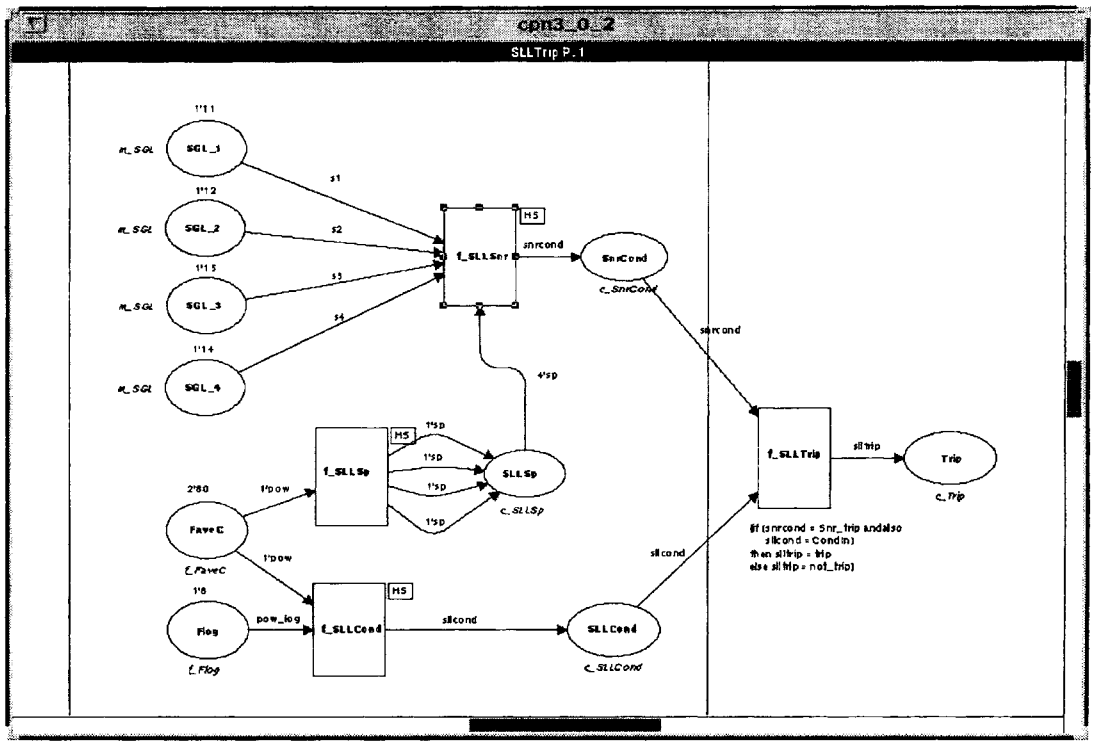

Fig. 2.1. CPN Model of SDS2 Steam Generator Low Level Trip Function

programming language with rich specification and simulation possibilities. The programming language through which CPN specifies desired operations in arc expressions and transition codes is ML.

In CPN, the most straightforward kind of analysis is simulation, which in many respects is similar to the testing and execution of a program. However, it is obvious that, by means of simulation, it is impossible to obtain a complete proof of dynamic properties of CPN. Therefore, there exist a number of formal analysis methods that are based on mathematical proof techniques. The analysis methods are occurrence graphs, place and transition invariants, and reduction rules techniques. The basic idea behind occurrence graphs is to construct a graph containing a node for each reachable marking and an arc for each occurring binding element. Obviously such a graph may become very large, even for small system. This problem can lead to state explosion. Therefore, we must simplify the CPN by omitting the cycle counters. As reduction methods of occurrence graphs, there exist symmetrical markings, stubborn sets, covering markings and proof rules methods. Place and transition invariants technique is to construct equations which are satisfied for all reachable marking. Analysis by means of place and transition invariants has several attractive properties. First, it is possible to obtain an invariant for a hierarchical CPN by composing invariants of the individual pages. Secondly, we can find invariants without fixing the system parameters, and hence we can

obtain general properties which are independent of the system parameters. Thirdly, we can construct the invariants during the design of a system and this will usually lead to an improved design. However, the main drawback of invariants analysis is the fact that it requires skills which are considerably higher and more mathematical than those required by the other analysis methods. Finally, CPN can also be analyzed by means of reduction, where the basic idea is as follows: First we choose one or more types of properties which we want to investigate. Then we define a set of reduction rules by which we can simplify CPN - without changing those properties which we are investigating. A serious problem for many reduction methods is that the absence of a property in the reduced net does not tell much about why the property is absent in the original net [2].

One example Design/CPN model is shown in Fig. 2.1. In this Figure, the CPN model represents the Wolsung NPP SDS2 function (Steam Generator Low Level Trip).

2.2. Prototype Verification System

PVS is a verification system: an interactive environment for writing formal specifications and checking formal proofs. It builds on nearly 20 years experience at SRI in building verification systems, and on substantial experience with other system. The distinguishing feature of PVS is a synergistic integration of an expressive specification language and powerful theorem-proving capabilities. PVS has been applied successfully to large and difficult applications in both academic and industrial settings.

PVS provides an expressive specification language that augments classical higher-order logic with a sophisticated type system containing predicate subtypes and dependent types, and with

```
1: sum : THEORY
2: BEGIN
3:
4:    n: VAR nat
5:    sum(n): RECURSIVE nat =
6:          (IF n=0 THEN 0 ELSE n + sum(n-1) ENDIF)
7:          MEASURE (LAMBDA n: n)
9:
10:   closed_form: THEOREM sum(n) = (n*(n+1)/2
11:
12: END sum
```

Fig. 2.2. Specification Language by PVS

parameterized theories and a mechanism for defining abstract data types such as lists and trees. The standard PVS types include numbers (reals, rationals, integers, naturals, and the ordinals to ϵ_0), records, tuples, arrays, functions, sets, sequences, lists, and trees, etc. The combination of features in the PVS type-system is very convenient for specification, but it makes typechecking undecidable. The PVS typechecker copes with this undecidability by generating proof obligations for the PVS theorem prover. Most such proof obligations can be discharged automatically.

PVS has a powerful interactive theorem prover/proof checker. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The PVS proof checker manages the proof construction process by prompting the user for a suitable command for a given subgoal. The execution of the given command can either generate further subgoals or complete a subgoal and move the control over to the next subgoal in a proof. User-defined proof strategies can be used to enhance the automation in the proof checker. Model-checking capabilities

```
Closed_form :
   |-----
{1}   (FORALL (n: nat): sum(n) = (n*(n+1)) /2 )

Rule? (induct "n")
Inducting on n,
This yields 2 subgoals:
Closed_form.1:
   |-----
{1}      sum(0) = (0*(0+1)) / 2

Rule? (postpone)
Postponing closed_form.1.

Closed_form.2 :
|----------
{1}      (FORALL (j:nat):
          sum(j) - (j*(j+1)) /2
                    IMPLIES sum(j+1) = ((j+1) * (j+1) * (j+1+1)) /2
                                  .
                                  .
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of closed_form.2.
Q.E.D.
```

Fig. 2.3. PVS Proof System

used for automatically verifying temporal properties of finite-state systems have recently been integrated into PVS. PVS's automation suffices to prove many straightforward results automatically: for hard proofs, the automation takes care of the details and frees the user to concentrate on directing the key steps [5].

One example of PVS specification language is shown in Fig. 2.2. Fig. 2.3 shows a part of PVS proof system.

3. Verification Technique by Relating CPN with PVS

It is generally well accepted that the complete analysis of requirements is critical for safety and effectiveness of the system. In this work, we have selected CPN as a modeling tool. As mentioned in section 2, in order to analyze requirements of the system based on CPN models, various invariants analysis and reachability analysis techniques have been proposed in many researches [2]. However, these techniques have their own limitations that they can consider only the behavioral aspects of the models, not deal with all the logical aspects, and easily lead to the state explosion problem.

Therefore, we suggest an axiom or lemma based-analysis method in order to supplement the typical CPN analysis methods. For the basis of the axiom or lemma based-analysis, we have shown that CPN models could be translated into axioms or lemmas in a straightforward manner. The translation process is automated through the extractor and the translator developed in this work.

In order to realize the automated analysis for the axioms or lemmas translated from CPN models, an effective theorem prover is required. PVS, which is well known as the verification tool, has been selected in this work. Fig. 3.1 depicts the automatic software verification tool schematically.

3.1. CPN Modeling

For modeling systems with CPN, first of all, complete analysis of the system is needed since an error in the requirements may result in serious faults of software. The graphical CPN model may help us to analyze, visualize and simulate system requirements. In CPN, requirements are modeled by objects such as Port, Arc, Place and Transition. Then, CPN model gives users the effects of the rapid prototyping and the visualizing of requirements. Therefore, it is very easy for developers to exchange opinions with users by CPN model.

As shown in Fig. 3.1, CPN model has advantages to analyze the correctness of requirements with type check, syntactic and semantic validation, that is, the simulation with CPN model may prove the completeness of

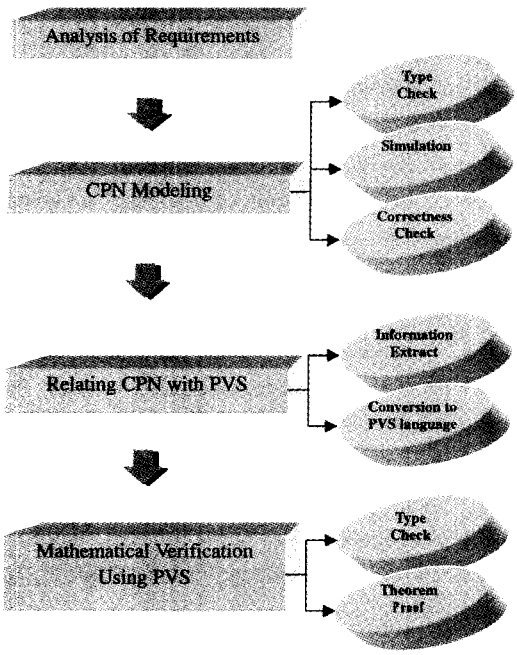

Fig. 3.1. Verification Scheme

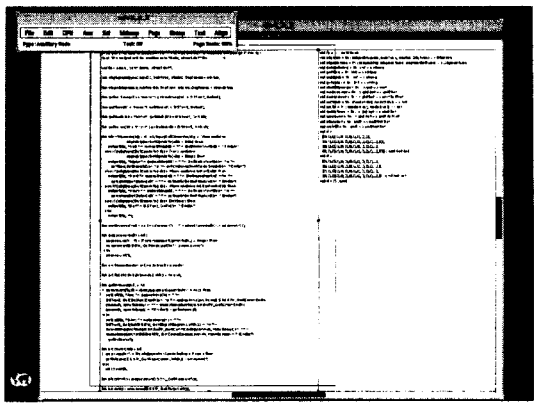

Fig. 3.2. A Process of the Extractor in Design/CPN

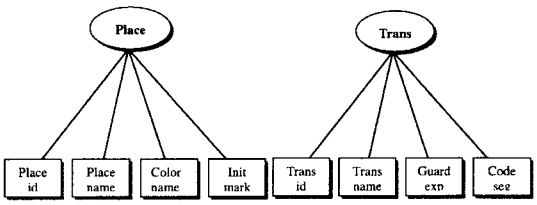

Fig. 3.3. The Example Structure of Extracted Information

requirements and the consistency with requirements. In Design/CPN, type checking is performed by 'Syntax Check' command. Type check supports not only syntactic check of data type but check for properties such as completeness and consistency. 'Enter Simulator' and 'Interactive Run' commands show the feature of simulation process along the each path. Specially, it is shown path by path that tokens of the each page move around. As a result of type check and simulation, correctness check of requirements has been achieved successfully. However, CPN is not proper for the mathematical verification of the system. In this work, therefore, PVS is used for the mathematical verification. Then, PVS help CPN for analysis of the system requirement. PVS also supports the type check function. Data type mismatching is checked by type check function in PVS. In the process of Theorem Proofs, semantic check of PVS specification from system requirement is performed and that is checked whether PVS specification is correct logically or not. As a result of this process, software is verified mathematically and may have the enhanced reliability.

3.2. Conversion to PVS Specification Language Using Extractor and Translator

It is not meaningless to perform an analysis for each model independently because the analysis of the system is performed in the "divide and conquer" manner. In Design/CPN, it is impossible to access to a data structure of the system directly, but the system information can be extracted by using a ML program. That is, Design/CPN has

```
begin
read the data structure from the output of extractor;
do {
    find a transition t;
    if ( t has a guard or arc expression ) then
        { check whether clauses can be used for AXIOM or LEMMA;
          if ( check = OK ) then
                        { find input;
                        if ( there exists input place that is the output place
                            of another transition, and the transition has the
                            OK guard or arc expression)
                        then translate the guard or arc expression into LEMMA;
                        else translate the guard or arc expression into AXIOM;
                        }
          else PASS;
        }
    else PASS;
} until ( there is no more guard or arc expression );
Let the final (i.e., last linked to output place) guard or arc expression
        be THEOREM;
end.
```

Fig. 3.4. The Algorithm of the Translator

internal functions for asking questions to modeled system. Using this ML functions and programs, we can extract the required system information from the CPN models. Therefore, an extractor has been used in this work. The extractor is run on the Design/CPN, as shown in Fig. 3.2. Objects for converting in CPN are Place, Port, Transition, Sub-Transition, Arc, Color, and variable declaration. When the extractor has met Places in CPN models, that extracts the information such as place id, place name, color name and initial marking. And the extractor can extract transition id, transition name, guard expression and code segment in Transition objects. Fig. 3.3 shows the

example structure of extracted information.

In PVS, it is hard to verify a system with the above results. Therefore, we need a process that the results from CPN model are converted to PVS inputs (PVS specification language). This process is performed with the translator developed in this work. The translator is also constructed with ML language. The translation process is straightforward. Guards of a transition are translated to THEOREM, AXIOM or LEMMA of PVS, if clauses can be used both in guard regions of CPN and for THEOREM, AXIOM or LEMMA expressions. For arc expressions, the same rules are applied. Fig. 3.4 describes the algorithm that the guards and arc expressions of CPN are translated into AXIOMs or LEMMAs of PVS.

When the translation process has finished successfully, PVS input form such as Fig. 2.2 is generated and we can verify with this converted input mathematically.

3.3. Verification and Validation Using PVS

In this work, the method of verification is the mathematical verification for the converted PVS specification language. PVS specification language for each page is represented with timed states. Thus, we have considered the signal and the power of the system as functions of time. Then, it is proved mathematically using the PVS proof system.

PVS proof system has interactive form with the real examiner. Therefore, in PVS proof system, prompt is [Rule?]. In this work, PVS proof system analyzes a specification language to be based on conditional sentences of transition, and, we investigate that these conditional sentences are realized in all-time cases. In addition, PVS proof system shows the proof case at constant time(t=0). In PVS proof system, proof commands like skosimp(Skolemize then Flatten), prop (Propositional Simplification), postpone(Go to Next Remaining Goal), grind(induct and simplify) are used.

3.4. Discussion

Requirements phase V&V tasks defined in IEEE std. 1012 are as follows:

(1) Software Requirements Traceability Analysis. : Trace SRS (Software Requirement Specification) requirements to system requirements in concept documentation. Analyze identified relationships for correctness, consistency, completeness, accuracy.

(2) Software Requirement Evaluation. : Evaluate SRS requirements for correctness, consistency, completeness, accuracy, readability, and testability. Assess how well SRS satisfies software system objectives. Assess the criticality of requirements to identify key performance or critical areas of software.

(3) Software Requirements Interface Analysis. : Evaluate SRS with hardware, user, operator, and software interface requirements documentation for correctness, consistency, completeness, accuracy, and readability.

(4a) Test Plan Generation. : Plan system testing to determine if software satisfies system objectives. Criteria for this determination are, at a minimum: (a) compliance with all functional requirements as complete software end item in system environment (b) performance at hardware, software, user, and operator interfaces (c) adequacy of user documentation (d) performance at boundaries (for example, data, interface) and under stress conditions. Plan tracing of system end-item requirements to test design, cases, procedures, and execution results. Plan documentation of test tasks and results.

(4b) Acceptance Test Plan Generation. : Plan

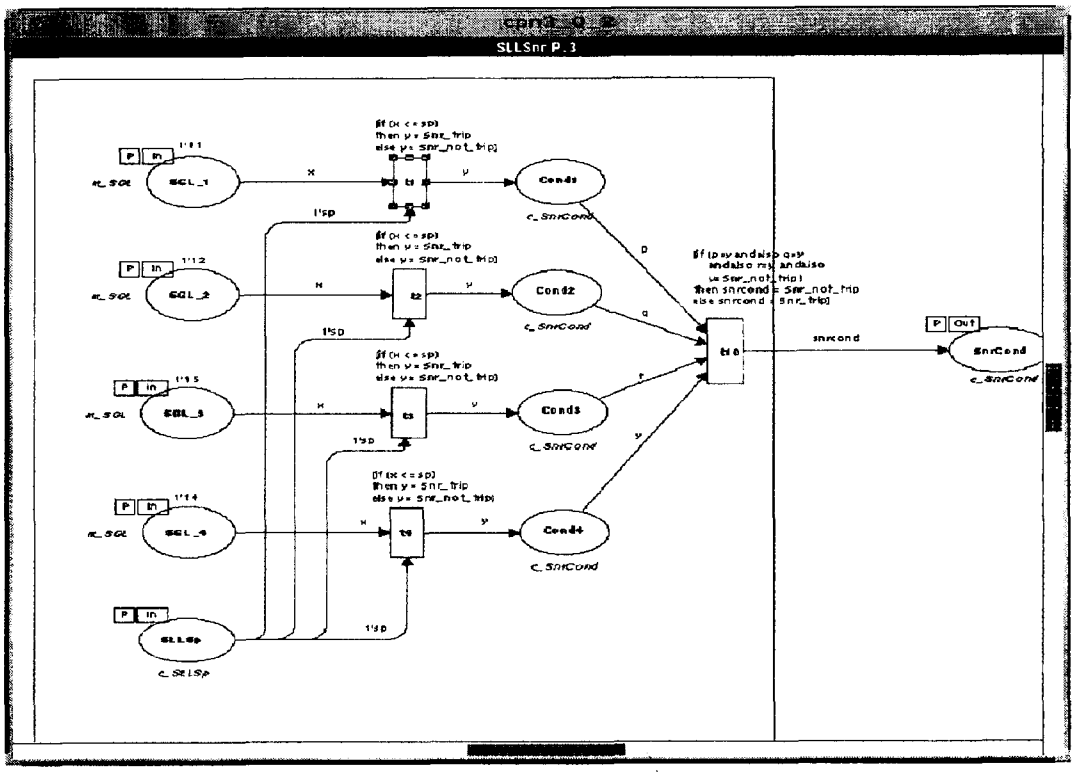

Fig. 4.1. CPN Model of f_SLLSnr Function

acceptance testing to determine if software correctly implements system and software requirements in an operational environment. Criteria for this determination are, at a minimum: (a) compliance with acceptance requirements in operational environment (b) adequacy of user documentation. Plan tracing of acceptance test requirements to test design, cases, procedures, and execution results. Plan documentation of test tasks and results.

Our work could deal with the issue (2), (3) and (4a) and it is difficult to handle (1) and (4b) through the method proposed in this work. In the CPN modeling process, consistency, accuracy and readability of system requirements could be addressed through the various analysis methods that Design/CPN has offered basically. The CPN model could be simulated and analyzed in view of occurrence graphs with ease in Design/CPN. This may prove the completeness of requirements and the consistency with requirements. In system test plan generation activity, the information through the simulation of Design/CPN can help to test design, cases, procedures, and execution results. On the other hand, with PVS, we can find the fact that no significant problem can be found logically and we can evaluate completeness and correctness of requirements. In this way, our work can address some issues of requirements V&V phases partially or entirely.

In addition, it is the state of the art of software V&V that the commercial automated-V&V tool

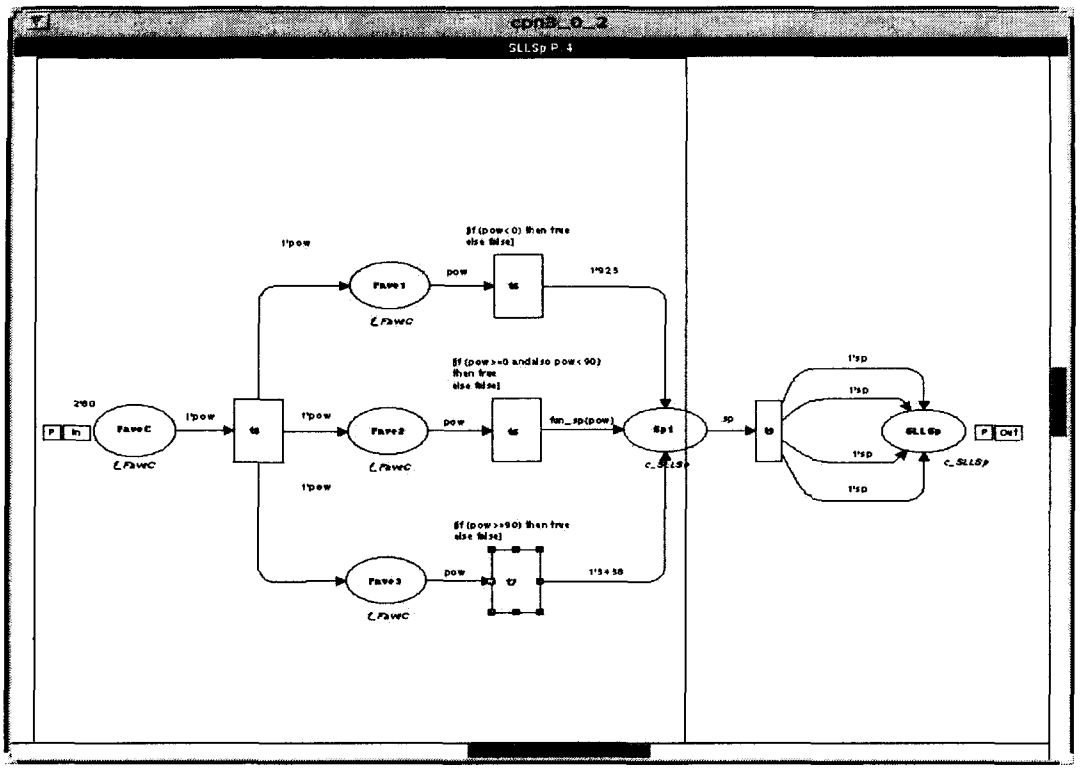

Fig. 4.2. CPN Model of f_SLLSp Function

does not exist yet. And another modeling tool such as CPN in this work is the Statecharts [1]. Statecharts are very similar to CPN as a point of expressive power and used to specify state transitions in reactive system. However, Statecharts don't have automated formal analysis methods.

4. Application

4.1. CPN Modeling

In this work, the target requirement is that for the Wolsung NPP SDS2 function (Steam Generator Low Level Trip). In this requirement, the functional requirements such as f_SLLCond, f_SLLCondA, f_SLLSnr, f_SLLSp, f_SLLSpD, and f_SLLTrip are included. These functional requirements are modeled with Design/CPN.

Fig. 2.1 shows the top-level CPN model of the function, and other sub-models are shown in Fig. 4.1 to 4.3

4.2. Conversion to PVS Specification Language Using ML Language

This section is for the process to help flow-through from requirements to PVS specification language using extractor and translator.

As mentioned in section 2.1, PVS specification language is composed of simple THEORYs. Therefore, the main page is translated to SLLTrip

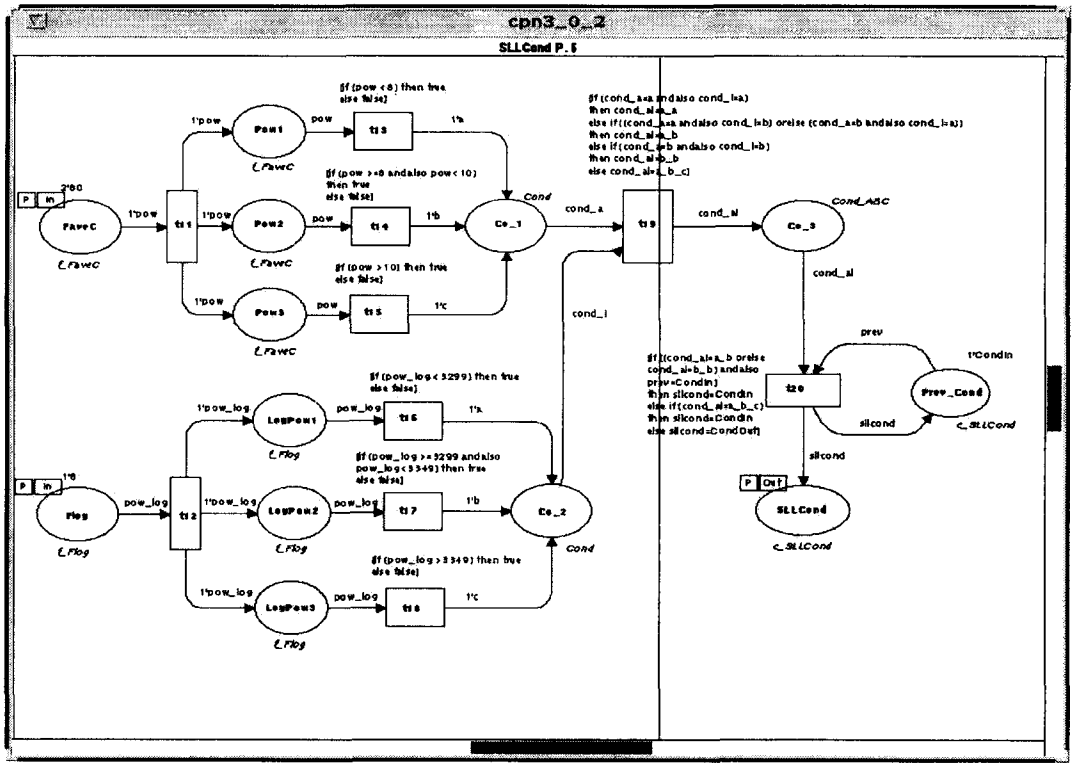

Fig. 4.3. CPN Model of f_SLLCond Function

function THEORY and sub-pages to SLLSnr, SLLSp and SLLCond function THEORYs, respectively. Each THEORY is described in the next section.

4.3. Verification and Validation Using PVS

This section describes PVS specification converted from SDS2 system models by the translator. The PVS specification language is composed of SLLTrip.pvs (Fig. 4.7), SLLSnr.pvs (Fig. 4.8), SLLSp.pvs (Fig. 4.9), and SLLCond.pvs (Fig. 4.10) files, named after CPN model pages. The declaration part is translated to dictionary.pvs (Fig. 4.5) and variables.pvs (Fig. 4.6) files which are imported to the THEORYs using IMPORTING command.

Fig. 4.11 describes the partial of proof in SLLSnr specfication language.

4.4. Discussion

In this application for Wolsung NPP SDS2, system requirements were modeled by CPN. Therefore, rapid prototyping and visualization could be achieved and it was very easy to simulate the system. In Design/CPN, the correctness and consistency of requirements could be analyzed through various analysis methods. As a result of this application, we could found that any logical problems do not exist in the specification. We also found that

```
dictionary  % [ parameters ]
                        : THEORY
  BEGIN
  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  time : TYPE = nat
  m_SGL : TYPE = int
  f_FaveC : TYPE = int
  f_Flog : TYPE = int
  c_SLLSp : TYPE = int
  c_SnrCond : TYPE =  Snr_trip,
Snr_not_trip
  c_SLLCond : TYPE =  CondIn,
CondOut
  c_Trip : TYPE =  trip, not_trip
  Cond_ABC : TYPE =  a_a, a_b,
b_b, a_b_c
  Co : TYPE =  a, b, c

  XT : TYPE = [time -> m_SGL]
  YT : TYPE = [time -> c_SnrCond]
  POWT : TYPE = [time -> f_FaveC]
  POWLOGT : TYPE = [time ->
f_Flog]

  END dictionary
```

Fig. 4.5. Specification Language of Dictionary.pvs

```
variables  % [ parameters ]
                        : THEORY
  BEGIN
  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  IMPORTING dictionary
  % variable declare

  t : time
  s1 : m_SGL
  s2 : m_SGL
  s3 : m_SGL
  s4 : m_SGL
  pow : POWT
  pow_log : POWLOGT
  sp : c_SLLSp
  snrcond : c_SnrCond
  sllcond : c_SLLCond
  prev : c_SLLCond
  slltrip : c_Trip
  x1 : XT
  x2 : XT
  x3 : XT
  x4 : XT
  y1 : YT
  y2 : YT
  y3 : YT
  y4 : YT
  cond_a : Co
  cond_l : Co
  cond_al : Cond_ABC

  END variables
```

Fig. 4.6. Specification Language of Variables.pvs

```
SLLTrip  % [ parameters ]
                        : THEORY
  BEGIN
  % ASSUMING
   % assuming declarations
  % ENDASSUMING

  IMPORTING dictionary
  IMPORTING variables
  IMPORTING SLLSnr
  IMPORTING SLLCond
  IMPORTING SLLSp

  f_SLLTrip  :  THEOREM  (IF
snrcond=Snr_trip AND sllcond=CondIn
THEN slltrip=trip ELSE slltrip=not_trip
ENDIF)

  END SLLTrip
```

Fig. 4.7. Specification Language of SLLTrip.pvs

CPN has insufficient expressive power to specify time-related properties. Therefore we can suggest improving the CPN.

Furthermore, with PVS, the mathematical verification for CPN models was performed in order to achieve the improved software reliability. In order to enhance the mathematical verification with PVS, more proof strategies of

```
SLLSnr  % [ parameters ]
                        : THEORY
  BEGIN
  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  IMPORTING dictionary
  IMPORTING variables

  t1 : AXIOM (IF x1(t) <= sp THEN
y1(t)=Snr_trip ELSE y1(t)=Snr_not_trip
ENDIF)
  t2 : AXIOM (IF x2(t) <= sp THEN
y2(t)=Snr_trip ELSE y2(t)=Snr_not_trip
ENDIF)
  t3 : AXIOM (IF x3(t) <= sp THEN
y3(t)=Snr_trip ELSE y3(t)=Snr_not_trip
ENDIF)
  t4 : AXIOM (IF x4(t) <= sp THEN
y4(t)=Snr_trip ELSE y4(t)=Snr_not_trip
ENDIF)
  t10 : THEOREM
        (FORALL(t:time):(IF
        y1(t)=y2(t) AND y2(t)=y3(t)
        AND y3(t)=y4(t) AND
        y4(t)=Snr_not_trip THEN
        snrcond=Snr_not_trip ELSE
        snrcond = Snr_trip ENDIF))

  END SLLSnr
```

Fig. 4.8. Specification Language of SLLSnr.pvs

```
SLLSp  % [ parameters ]
                        : THEORY
  BEGIN
  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  IMPORTING dictionary
  IMPORTING variables

  t8 : THEOREM
(FORALL(t:time):( IF pow(t) < 0 THEN
sp=923 ELSE (IF pow(t)>=0 AND
pow(t)<90 THEN sp=(28*pow(t)+923)
ELSE sp=3438 ENDIF) ENDIF))

  END SLLSp
```

Fig. 4.9. Specification Language of SLLSp.pvs

```
SLLCond  % [ parameters ]
                        : THEORY
  BEGIN
  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  IMPORTING dictionary
  IMPORTING variables

  T11 : THEOREM
(FORALL(t:time):(IF pow(t)<8 THEN
cond_a=a ELSE (IF pow(t)>=8 AND
pow(t)<10 THEN cond_a=b ELSE
cond_a=c ENDIF) ENDIF))

  T12 : THEOREM
(FORALL(t:time):(IF pow_log(t)<3299
THEN cond_l=a ELSE (IF
pow_log(t)>=3299 AND
pow_log(t)<3349 THEN cond_l=b
ELSE cond_l=c ENDIF) ENDIF))

  T19 : THEOREM (IF cond_a=a AND
cond_l=a THEN cond_al=a_a ELSE (IF
(cond_a=a AND cond_l=b) OR
(cond_a=b AND cond_l=a) THEN
cond_al=a_b ELSE (IF cond_a=b AND
cond_l=b THEN cond_al=b_b ELSE
cond_al=a_b_c ENDIF) ENDIF)
ENDIF)

  T20 : THEOREM (IF (cond_al=a_b
OR cond_al=b_b) AND prev=CondIn
THEN sllcond=CondIn ELSE (IF
cond_al=a_b_c THEN sllcond=CondIn
ELSE sllcond=CondOut ENDIF)
ENDIF)

  END SLLCond
```

Fig. 4.10. Specification Language of SLLTCond.pvs

```
t10:
  |------
1   (FORALL (t: time):
      (IF y1(t) = y2(t) AND y2(t) = y3(t) AND y3(t) = y4(t) AND y4(t) =Snr_not_trip
       THEN snrcond = Snr_not_trip ELSE snrcond = Snr_trip ENDIF))

Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
t10:
  |------
1  (IF y1(t!1) = y2(t!1) AND y2(t!1) = y3(t!1) AND y3(t!1) = y4(t!1)
     AND y4(t!1) =  Snr_not_trip  THEN snrcond = Snr_not_trip
     ELSE snrcond = Snr_trip
     ENDIF)

Rule? (prop)
Applying propositional simplification,
this yields  5 subgoals:
t10.1:
-1   y1(t!1) = y2(t!1)
-2   y2(t!1) = y3(t!1)
-3   y3(t!1) = y4(t!1)
-4   y4(t!1) = Snr_not_trip
  |------
1   snrcond = Snr_not_trip

Rule? (case "t!1=0")
Case splitting on
   t!1=0,
this yields  2 subgoals:
t10.1.1:
-1   t!1 = 0
[-2]   y1(t!1) = y2(t!1)
[-3]   y2(t!1) = y3(t!1)
[-4]   y3(t!1) = y4(t!1)
[-5]   y4(t!1) = Snr_not_trip
  |------
[1]   snrcond = Snr_not_trip
```

Fig. 4.11. The Partial of Proof in SLLSnr Specification

```
Rule? (grind)
stty: : 303770706510 00167466
stty: : 303770706510 00167466
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:

t10.1.1:
[-1]   t!1 = 0
-2   y1(0) = y4(0)
-3   y2(0) = y4(0)
-4   y3(0) = y4(0)
-5   Snr_not_trip?(y4(0))
  |-------
1   Snr_not_trip?(snrcond)

Rule? (postpone)
Postponing t10.1.1.
t10.1.2:
[-1]   y1(t!1) = y2(t!1)
[-2]   y2(t!1) = y3(t!1)
[-3]   y3(t!1) = y4(t!1)
[-4]   y4(t!1) = Snr_not_trip
  |-------
1   t!1 = 0
[2]   snrcond = Snr_not_trip

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
t10.1.2:
[-1]   y1(t!1) = y2(t!1)
[-2]   y2(t!1) = y3(t!1)
[-3]   y3(t!1) = y4(t!1)
[-4]   y4(t!1) = Snr_not_trip
  |-------
[1]   t!1 = 0
[2]   snrcond = Snr_not_trip
```

Fig. 4.11. The Partial of Proof in SLLSnr Specification (Cont'd)

PVS proof checker should be also developed.

5. Conclusions and Further Study

In this work, an axiom or lemma based-analysis method for CPN models is newly suggested in order to complement CPN analysis methods. Since an existing analysis techniques have their own limitations that they can consider only the behavioral aspects of the models, not deal with all the logical aspects, and easily lead to the state explosion problem. In addition, a guideline for the use of formal methods in order to apply them to NPP software V&V is

proposed in this work.

We could visualize the system requirements easily by using CPN model and verify the system mathematically by using PVS. The integration of CPN and PVS, which realizes rapid prototyping and mathematical verification, can enhance the software reliability by logical analysis using PVS. CPN and PVS can cover disadvantage of each other. In order to help flow-through from modeling by CPN to mathematical proof by PVS, an information extractor from CPN models has been developed. And a translator also has been developed and used in this work. Our software verification method is demonstrated to be useful with a simple example application and the completeness and consistency of system requirements is proved successfully.

In the future, we are planning to use this software verification method for safety critical system in other areas. As a result of this work, this software verification method may be a foundation of enhancement for Design/CPN and PVS. And, in order to improve the CPN, timing issue must be addressed and this can be based on an advancing state based modeling tool. In addition, we are going to develop the more adequate prover. Then we can realize the integrated software V&V tool.

References

1. J.M. Wing, "A specifier's introducion to formal method," *IEEE Computer*, **23**(9):8-24, (1990).
2. Kurt Jensen, "Coloured Petri Nets (Basic Concepts, Analysis Methods and Practical Use Volume 1), Second Edition", Springer-Verlag Berlin Heidelberg, (1997).
3. Tae-ho Kim, "Verification of Safety-Critical System Requirements using PVS", Master Thesis, Department of Computer Science, KAIST, (1997).
4. Jeffrey D. Ullman, "Elements of ML Programming", Prenice-Hall, Inc, (1994).
5. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, Mandayam Srivas, "A Tutorial Introduction to PVS", Computer Science Laboratory, SRI International, Updated June (1995).
6. S. Owre, N. Shankar and J. M. Rushby, "The PVS Specification Language(Beta Release)", Computer Science Laboratory, SRI International, April 12, (1993).
7. N. Shankar, S. Owre and J. M. Rushby, "The PVS Proof Checker: A Reference Manual(Beta Release)", Computer Science Laboratory, SRI International, March 31, (1993).
8. Sam Owre and John Rushby, "FME '96 Tutorial: An Introduction to Some Advanced Capabilities of PVS", Computer Science Laboratory, SRI International, (1996).
9. IEEE Standard-1012, "Software Verification and Validation Plans", IEEE, Inc, (1986).