Available online at www.sciencedirect.com
SCIENCE DIRECT
Reliability Engineering and System Safety 87 (2005) 351–363
RELIABILITY ENGINEERING & SYSTEM SAFETY
www.elsevier.com/locate/ress

Formal verification of functional properties of a SCR-style software requirements specification using PVS☆

Taeho Kim[a,b,*], David Stringer-Calvert[c], Sungdeok Cha[a]

[a]Computer Science Division, EECS Department and AITRC/SPIC/IIRTRC, Korea Advanced Institute of Science and Technology (KAIST), Daejon 305-701, Korea
[b]Digital Printing Division, Samsung Electronics, Suwon, Gyeonggi 443-742, Korea
[c]SRI International, Menlo Park, CA 94025, USA

Received 15 July 2003; accepted 25 June 2004
Available online 21 September 2004

Abstract

Industrial software companies developing safety-critical systems are required to use rigorous safety analysis techniques to demonstrate compliance to regulatory bodies. In this paper, we describe an approach to formal verification of functional properties of requirements for an embedded real-time software written in software cost reduction (SCR)-style language using PVS specification and verification system. Key contributions of the paper include development of an automated method of translating SCR-style requirements into PVS input language as well as identification of property templates often needed in verification. Using specification for a nuclear power plant system, currently in operation, we demonstrate how safety demonstration on requirements can be accomplished while taking advantage of assurance provided by formal methods.

Keywords: Safety-critical system; Software safety; Software requirements specification; Formal methods; Formal verification; Nuclear power plant system

1. Introduction

Safety-critical systems such as fly-by-wire aircraft and emergency shutdown systems for nuclear power plants are controlled by software, and regulation bodies routinely require rigorous safety demonstrations. Of all phases in software development, requirements engineering plays the most critical role in determining the overall software quality. According to NASA's data [13], nearly 75% of failures found in operational software are rooted in requirement errors. Among various approaches suggested for developing high-quality requirements specifications and conducting cost-effective analysis, formal methods are considered effective and promising [9]. We classify properties in requirements to be verified as either structural or functional properties. In Ref. [10], we demonstrated that PVS specification and verification system is useful in verifying SCR-style requirements for structural correctness such as consistency among input and output definitions and lack of circular reference and also Heitmeyer have researched on the similar problem [7]. As industrial systems are quite complex in that requirements document which often consists of several hundred pages long, the need for automated analysis is critical. In fact, Kim and Cha [10] demonstrated that much of tedious and potentially error-prone inspection process in the requirements can be fully automated and that inspection team could focus their effort on intellectually-challenging and domain-specific properties.

In this paper, we extend work reported in Ref. [10] to verify application-dependent properties which we refer to as functional properties. They are often stated in natural language and specify constraints to be satisfied by the system. Functional properties are often derived from the results of failure mode and effect analysis (FMEA) or domain expert's knowledge. In the case of Wolsung

☆ Preliminary version of this paper was presented at TACAS 2002 (Tools and Algorithms for the Construction and Analysis of Systems).

* Corresponding author. Address: Computer Science Division, EECS Department and AITRC/SPIC/IIRTRC, Korea Advanced Institute of Science and Technology (KAIST), Daejon 305-701, Korea.
E-mail addresses: thkim@salmosa.kaist.ac.kr (T. Kim), davesc@sri.com (D. Stringer-Calvert), cha@salmosa.kaist.ac.kr (S. Cha).

0951-8320/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ress.2004.06.005

PHT Low Core Differential Pressure (PDL)

1: The PHT Low Core Differential Pressure (ΔP) trip parameter includes both
2: an immediate and a delayed trip setpoint. Unlike other parameters, the ΔP
3: parameter immediate trip low power conditioning level can be selected by the
4: operator. A handswitch is connected to a D/I, and the operator can choose
5: between two predetermined low power conditioning levels.
6: The PHT Low Core Differential Pressure trip requirements are:
7: ……
8: e. Determine the immediate trip conditioning status from the conditioning level
9: D/I as follows:
10: 1. If the D/I is open, select the *0.3% FP* (Full Power) conditioning level.
11: If $\phi_{LOG} < 0.3\%FP\text{-}50\ mV$, condition out the immediate trip.
12: If $\phi_{LOG} >= 0.3\%FP$, enable the trip.
13: ……
14: g. If any ΔP signal is either below the immediate trip setpoint or above the
15: high irrational trip set point, open the appropriate loop trip error message D/O.
16: If no PHT ΔP delayed trip is pending or active then execute a delayed
17: trip as follows:
18: 1. Continue normal operation without opening the parameter trip D/O for
19: nominally three seconds.
20: 2. After the delay period has expired, open the parameter trip D/O
21: if f_{AVEC} equals or exceeds *80%FP*
22: Do not open the parameter trip D/O if f_{AVEC} is below *80%FP*
23: 3. Once the delayed parameter trip has occurred,
24: keep the parameter trip D/O open for one second,
25: and then close the parameter trip D/O once all DP signals are
26: above the delayed trip setpoint or f_{AVEC} is below *80%FP*
25: ……
24: h. Immediate trips and delayed trips (pending and active) can occur simultaneously.
25: ……

Fig. 1. Example of program functional specification.

shutdown system number 2 (SDS2), nuclear power plant shutdown system which we used as case-study in this paper, functional properties to be verified are stated in a document named program functional specification (PFS) written in natural language. Building on Ref. [10] which developed procedures for automatically translating software requirements written in SCR-style specification, we identify several functional property patterns often needed when verifying real-time software and describe PVS-based proof procedures. Even though we use a case-study found in nuclear industry, verification procedures we propose are general enough to requirements for real-time systems based on the Parnas' four-variable model [19] and synchronous language.

The Wolsung SDS2 is designed to continuously monitor the reactor state (e.g. temperature, pressure, and power) and to generate a trip (e.g. shutdown the plant and display an alarm) signal if the system enters unsafe state. The trip signal should be generated when the system detect unsafe system states by comparing critical parameter values such as pressure or temperature against the predefined threshold values. Software requirements specification (SRS), written in SCR-style notation, consists of about 200 pages of diagrams and tables, and the system-level requirements named PFS, authored by domain experts and written in English, is about 21 pages long. In this case-study, we concentrate on the trip condition named PDL_trip which is the most complex one among three trip conditions. SRS and PFS for PDL_trip is about 22 and four pages long, respectively. Safety constraints to be satisfied by the system are usually derived from PFS documents whose portions are shown in Fig. 1 (line numbers have been added for illustration purpose only).

SCR-style requirements [1] consist of four components. They are variable definitions, functional overview diagrams (FODs), structured decision tables (SDTs), and timing function definitions. Variable definitions describe interface between the computer system and its environment in terms of monitored and controlled variables. Monitored variables, with `m_`prefix, refer to the inputs to the computer system. Similarly, controlled variables refer to externally visible outputs generated to the environment via actuators. For each variable, attributes such as type, units, range, or accuracy are defined. FODs represent a hierarchical organization of functions using a notation similar to the data-flow diagram. A group, denoted by the `g_`prefix, consists of subgroups or basic functions, and function definition is given `f_`prefix. (Fig. 2). A function, whose definitions are captured in SDT, is assumed to take no time to compute the output value. The required behavior of each basic function is expressed in a tabular notation, called SDT, as shown in Fig. 3. For example, the output of the function `f_PDLCond`[1] is either `k_CondOut` or `k_CondIn` whose prefix indicates that

[1] We performed case-study on a safety-critical system currently in operation to demonstrate scalability as opposed using an artificial and toy-sized problem. Consequently, names used in Wolsung SDS2 are used in this paper without modification.

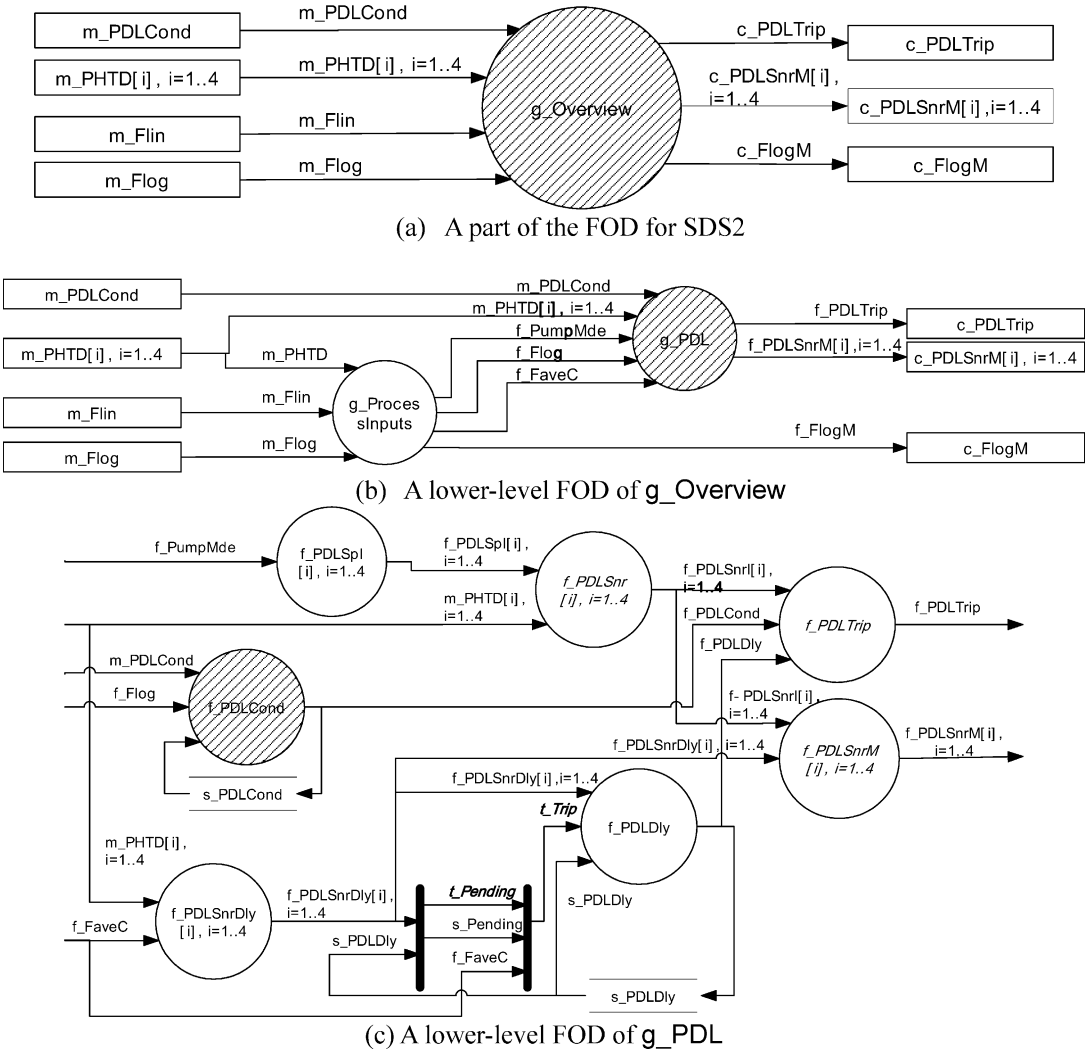

Fig. 2. Examples of the function overview diagram.

the output is a constant value. Condition macros are used as needed to enable concise specification of SDT. Fig. 3 indicates that the function returns the value `k_CondOut` when `m_PDLCond` is equal to `k_CondSwLo` and `w_FlogPDLCondLo[f_Flog]` is equal to `a`. The '–' entries denote the 'don't care' condition. In addition to graphically capturing input–output relations and required computations, state variables, with `s_`prefix, store the function output computed during the previous cycle so that its value may be used in the current cycle. Finally, timing functions are drawn as a bar (|), as `t_Pending` and `t_Trip` shown in Fig. 2(c) illustrates. FOD implicitly

1: **Condition Macros:**
2: w_FlogPDLCondLo[f_Flog]
3: a f_Flog < k_FlogPDLLo - k_CondHys
4: b f_FlogPDLLo - k_CondHys <= f_Flog < k_FlogPDLLo
5: c f_Flog >= k_FlogPDLLo

Structured Decision Table:

CONDITION STATEMENTS				
m_PDLCond = k_CondSwLo	T	T	T	T
w_FlogPDLCondLo[f_Flog]	a	b	b	c
s_PDLCond = k_CondOut	-	T	F	-
ACTION STATEMENTS				
f_PDLCond = k_CondOut	X	X		
f_PDLCond = k_CondIn			X	X

Fig. 3. The SDT for f_PDLCond.

specifies internal data dependencies among various components as well and dictates the proper order of computing a set of functions. For example, in Fig. 2(c), as the output of the `f_PDLSnrI[i]`, `i = 1,…,4` function is used as input to the `f_PDLTrip`, the latter may be invoked only when computation of the former is completed. This is the same concept used in synchronous data-flow languages such as LUSTRE [6].

Timing function is defined using generic timing function named `t_Wait` whose formal definition is shown below. Initial value, at time zero, is FALSE, and it stays true for `Time_value` period when the value of $C(t)$ changes from false to true.

$$\texttt{t\_Wait}(C(\texttt{t}), \texttt{Time\_value}, \texttt{tol}) = \begin{cases} \texttt{true} & \text{if there exists an instant in time, } \texttt{t\_s} \in [\texttt{t} - \texttt{Timer\_value}, \texttt{t}] \\ & \text{such that } C(\texttt{t\_s}) \text{ AND } \neg\, \texttt{t\_Wait}(C(\texttt{t\_s} - \varepsilon), \texttt{Time\_value}, \texttt{tol}) \\ \texttt{false} & \text{otherwise, including at } \texttt{t} = 0 \end{cases}$$

PVS is an interactive tool for writing specifications and constructing proofs [3]. It has been successfully used in several industrial applications. Examples [15–17,23] include verification of several communication and real-time protocols. One of the most complex systems verified by PVS to date is the AAMP5 microprocessor which has nearly half a million transistors [14]. PVS was also used to prove the completeness and consistency of conditions in the RSML specification of the TCAS II specification [8], and to prove an avionics control system [4]. Our decision to use PVS in performing verification of functional properties is based on the following factors:

1. PVS, a freely available software, is useful when developing an integrated safety analysis environment in which proofs of structural, functional, and safety properties are performed. Verification of structural properties has already been done in Ref. [10] using PVS, and the current release of PVS includes built-in support for SDT notations as well as automated analysis for consistency and completeness.
2. PVS specifications support an extension of classical higher-order logic with strong typing. Type checking helps to find errors in an early stage.
3. PVS theorem prover is based on the sequent calculus. It supports highly automated proof strategies as well as top-down proof exploration and construction

The rest of our paper is organized as follows. Section 2 describes the proposed verification procedure for functional properties, and Section 3 discusses results of our case-study and compares the proposed approach against other approaches. Section 4 concludes this paper.

2. Verification of SCR-style SRS

Our approach to verification of functional properties is conducted in the following steps:

1. *Editing SCR-style SRS and translate it to PVS*. An SRS editor and translator were provided, and the specification is translated into PVS specifications. Fig. 4 is a screen dump of the tool. This procedure can be

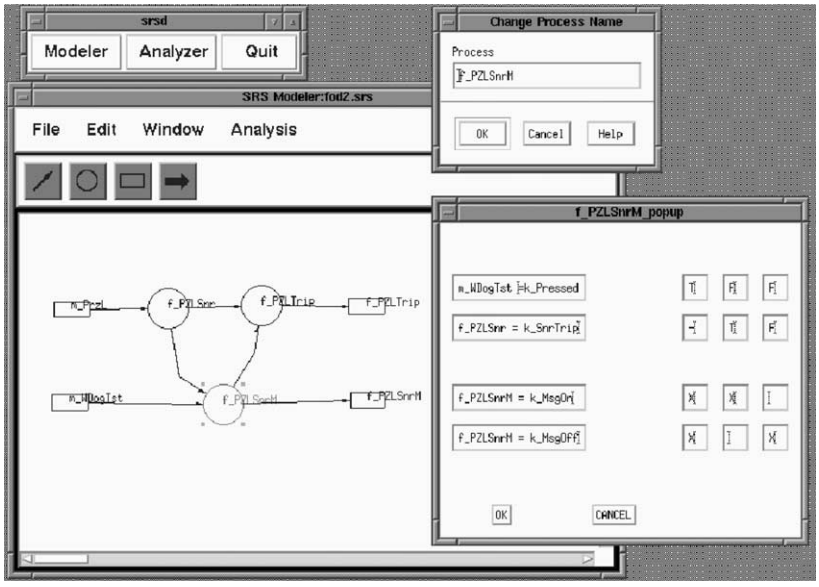

Fig. 4. An SCR-style SRS editor generating a PVS specification.

automated and the detailed procedure is described in Section 2.1.

2. *Translation of PFS into PVS*. Functional properties are derived from PFS in a natural language, English in case of Wolsung SDS2 SRS, and they are expressed as PVS theorems. Although this process is not automated, as English documents are used as inputs, cross-references and templates proposed in this paper assist translation process as explained in Section 2.2.
3. *Proof*. PVS software reads in requirements and properties to be verified, and a proof session is started. The later part of Section 2.2 explains how earlier proofs can be reused when proving similar properties.

2.1. Translation from SCR-style SRS to PVS

PVS input derived from SCR-style SRS through the following five steps: 1. definition of time (tick) model elements, 2. definition of types and constants, 3. definitions of types for monitored and controlled variables, 4. translation of SDTs, and 5. definition and translation of timing functions.

A unit of PVS specification is `THEORY`. The translated specifications for the Wolsung SDS2 PDL Trip consist of mainly three theories. The `THEORY`s are `THEORY tick`, `THEORY definition`, `THEORY time`, and `THEORY pdl`, and these `THEORY`s are stored in files tick.pvs, definition.pvs, time.pvs, and pdl.pvs, respectively. The `THEORY tick` defines a basic tick definition shown in step 1. The `THEORY definition` defines types, constants, and variables with importing `THEORY tick` and is translated in step 2, and 3. The `THEORY time` defines a generic timing function with importing `THEORY tick` and `THEORY definition`, and is translated in step 5(1). The `THEORY pdl` defines software functions in SRS with importing `THEORY tick`, `THEORY definition`, and `THEORY time`, and is translated in step 4 and 5(2). These theory templates are shown in Fig. 5(a)–(d). The numbering on the left is merely a line number for reference in this paper, and is not part of the translation procedure or translated specification. The `IMPORTING` keyword is used for including other theory such that definitions in imported theory could be used.

```
1: tick : THEORY
2:   BEGIN
3:     inserted by step 1: Definition of time model elements
4:   END tick
```

(a) `THEORY tick`

```
1: definition : THEORY
2:   BEGIN
3:            IMPORTING tick
4:     %      translated by step 2: Definition of types and constants
5:     %      translated by step 3: Definition of types
6:     %                    for monitored and controlled variable
7:   END definition
```

(b) `THEORY definition`

```
1: time : THEORY
2:   BEGIN
3:            IMPORTING tick
4:            IMPORTING definition
5:     %      translated by step 5(1): Definition of timing function
6:   END time
```

(c) `THEORY time`

```
1: pdl : THEORY
2:   BEGIN
3:            IMPORTING tick
4:            IMPORTING definition
5:            IMPORTING time
6:     %      translated by step 4: Translation of SDT
7:     %      translated by step 5 (2): Translation of timing function
8:   END pdl
```

(d) `THEORY pdl`

Fig. 5. Outlines of translated PVS specification.

```
1: tick  : TYPE+ = NAT CONTAINING 0
2: t     : VAR tick
3: init  : tick = 0
```

Fig. 6. Step 1: definition of model elements.

2.1.1. Step 1. Definition of time model elements

Time model must first be defined as a PVS theory if there are real-time requirements. Time increases by a fixed period, a positive number, so we specify time using a `tick`. The `tick` is a natural number and a time can be computed by multiplying `tick` and a fixed period between `tick`. This part is common through different specifications and is denoted in Fig. 6. Line 1 defines `tick` and line 2 defines `t`, representing a variable of type `tick`. In line 3, a constant `init` is defined to be 0, for using as the initial value of `tick`.

2.1.2. Step 2. Definition of types and constants

Once time model is defined, other data types and constant values need to be defined. If needed, data types defined in another theory file can be imported to avoid repetition. Analog variables are declared as real type whereas digital variables are declared as enumeration type. The values of variables with time are declared as functions from `tick` to the variable type. Fig. 7 shows the types and constant definitions used in the Wolsung SDS2. Line 1 shows the definition of `millivolt`, defined in the SCR-style as an analog variable, so it is translated to the `real` type. Line 2 is a definition of `t_Millivolt` as a function from `tick` to `millivolt`. Line 4 is a definition of the `zero_one` type for a digital variable, defined as set type whose membership includes 0 and 1. In line 5, `undef` is used to declare constants whose values are left undefined during requirements engineering phase but to be decided later during software development. `k_Trip` and `k_NotTrip` in lines 6 and 7 are constants of the digital variable type. Line 11 defines `to_TripNotTrip` as an enumeration of `k_Trip` and `k_NotTrip`. Lines 12 and 13 define a function `t_TripNotTrip` from tick to `to_TripNotTrip`. This type includes a trivial function mapping from any `tick` value `t` to the constant `k_Trip` for ensuring non-emptiness. The `to_CondInOut` is an enumeration type whose members are `k_CondIn` and `k_CondOut`. Line 15 is a function `t_CondInOut` from `tick` to `to_CondInOut`. Line 17 defines `enumabc` used within SDT. `enumabc` is an enumerative type for `a, b`, and `c`.

2.1.3. Step 3. Definition of types for monitored and controlled variable

This step defines the types of the monitored and controlled variables using the definitions from step 2. The variables are defined in the form `variable:type`. Fig. 8 is an example for monitored variable `m_Flog` and controlled variable `c_PDLTrip.m_Flog` is a type `t_Milivolt` in line 1 and `c_PDLTrip` is a type `t_TripNotTrip.t_Milivolt` is defined at line 2 in Fig. 7, and `t_TripNotTrip` is defined at line 12 in Fig. 7.

2.1.4. Step 4. Translation of SDTs

Definitions on time model and data types are used in further defining internal computations as either SDT or timing function. Hierarchical information is unnecessary when verifying correctness with respect to functional properties. The translation order of functions should be partial ordered, because of dependencies of definitions. There are two function types in SCR-style requirements depending on whether or not values of state variables are used.

Let *f_output*, *f_input1*, *f_input2*, and *s_output* be function names or variable names.

The first kind of function is

$$f\_output(t) = compute(f\_input1(t), f\_input2(t))$$

```
 1: millivolt : TYPE = real                                  % analog variable
 2: t_Millivolt : TYPE = [tick -> millivolt]
 3:
 4: zero_one : TYPE+ = {x:int | x=0 OR x=1} CONTAINING 0  % digital variable
 5: undef : TYPE+                                    % undefined-value constant
 6: k_Trip : zero_one = 0
 7: k_NotTrip : zero_one = 1
 8: k_CondIn : undef
 9: k_CondOut : undef
10:
11: to_TripNotTrip : TYPE = {x:zero_one | x = k_Trip OR x = k_NotTrip}
12: t_TripNotTrip : TYPE+ = [tick -> to_TripNotTrip]    % function type from
13:     CONTAINING lambda (t:tick) : k_Trip             % tick to_TripNotTrip
14: to_CondInOut : TYPE = {k_CondIn, k_CondOut}         % including. t->k_Trip
15: t_CondInOut : TYPE = [tick -> to_CondInOut]
16:
17: enumabc : TYPE = {a,b,c}
```

Fig. 7. Step 2: definition of types and constants.

```
1: m_Flog : t_Milivolt            % Type definition for monitored variable
2: c_PDLTrip : t_TripNotTrip      % Type definition for controlled variable
```

Fig. 8. Step 3: definition of types for monitored and controlled variables.

To compute *f_output*, it reads the values of the *f_input1* and *f_input2* at tick *t* and then compute *f_output* at tick *t*. For this function, the translation template is

```
1: f_output(t):value_type=
2:     compute(f_input1(t), f_input2(t))
```

If the condition macro is defined within compute, the macro should be locally defined by the LET…IN construct In this case, the translation template is[2]:

```
1: f_output(t:tick) : value_type =
2:     LET
3:       w_condition_macro : enumeration_type = condition_macro
4:     IN
5:       compute(f_input1(t), f_input2(t))
```

The second kind of function is

$$f\_output(t) = compute(f\_input1(t), s\_output(t))$$

$$s\_output(t) = \begin{cases} \text{initial\_value} & \text{when } t = 0 \\ f\_output(t-1) & \text{when } t \neq 0 \end{cases}$$

One must note that there exists a circular dependency in the above definition (e g. *f_output* and *s_output*) and that type checking mechanism of PVS does not allow circular dependencies in an explicit manner. However, one can use definitional style of function definition where local definitions are embedded within a recursive definition. That is, *f_output* is written as a finite state machine which refers previous state values (i.e. *s_output* in a local definition LET…*s_output*…IN) as well as the current input values to determine *f_output* value.

```
 1: f_output(t:tick) : RECURSIVE value_type =
 2:             LET
 3:               s_output: value_type =
 4:                    IF t = 0 THEN initial_value
 5:                    ELSE f_output(t-1)
 6:                    ENDIF
 7:             IN
 8:             output(f_input1(t), s_output)
 9:           MEASURE t
10: s_output(t:tick) : value_type = IF t = 0 THEN initial_value
11:                                 ELSE f_output(t-1)
12:                                 ENDIF
```

[2] In SCR-style SRS, functions and condition macros are defined as tabular notation, so w_condition_macro and computes in translated PVS specification are expressed as a TABLE…ENDTABLE construct.

The definition of f_output is given in lines 1–9. Line 8 refers to s_output, but as s_output is not defined until lines 10–12, so a local definition of s_output is given within the function f_output at lines 3–6. The keyword RECURSIVE is used to indicate a recursive function, and a MEASURE function provided to allow the type checker to generate proof obligations to show termination. The s_output at line 10–12 is a definition of the previous value of f_output.

Fig. 9 shows how SDT for PDLCond trip condition is translated, shown earlier in Fig. 3, as PVS function supports definition of primitive SDT constructs [18] as shown in lines 17–25. It is worth noting that in the PVS specification, the structure is nearly identical except the fact that rows and columns are transposed and that disciplined use of comments (or comment lines such as lines 17 and 19) further improves readability of PVS specification. While it is possible to express SDT in PVS in a different style using AXIOM construct so that local definitions are unneeded, step-by-step proof might be required for safety auditing.

2.1.5. Step 5. Definition and translation of timing function

The semantics of timing functions are defined as shown in Fig. 10. The function twf at lines 1–7 defines the output as FALSE when tick t = 0 and TRUE for a specified time interval tv after triggering a condition to TRUE (i.e. that ts is a current tick, the output at ts − 1 is FALSE, and the condition at ts is TRUE). The function twfs at lines 9–10 specifies a function from tick to an output(bool) to specify a trajectory of the function twf.

```
 1:  f_PDLCond(t:tick) : RECURSIVE to_CondInOut =
 2:    LET
 3:     s_PDLCond : t_CondInOut = IF t=0 THEN k_CondIn
 4:                              ELSE f_PDLCond(t-1)
 5:                              ENDIF,
 6:     w_FlogPDLCondLo : enumabc  =  TABLE
 7:        ...                       % similar to if-then-else
 8:        ENDTABLE,
 9:      X = (LAMBDA (x1: pred[bool]),
10:                  (x2: pred[enumabc]),
11:                  (x4: pred[bool]) :
12:         x1( m_PDLCond(t) = k_CondSwLo) &
13:         x2(     w_FlogPDLCondLo) &
14:         x3(           s_PDLCond = k_CondOut))  IN TABLE
15:        %     |    |     |
16:        %     v    v     v
17:        %-------|----|----|------------%
18:        | X( T  , a? , ~ )| k_CondOut ||
19:        %-------|----|----|------------%
20:        | X( T  , b? , T )| k_CondOut ||
21:        %-------|----|----|------------%
22:        | X( T  , b? , F )| k_CondIn  ||
23:        %-------|----|----|------------%
24:        | X( T  , c? , ~ )| k_CondIn  ||
25:        %-------|----|----|------------%
26:        ENDTABLE
27:  MEASURE t
28:
29:  s_PDLCond(t:tick):to_CondInOut =  IF t = 0 THEN k_CondIn
30:                                   ELSE f_PDLCond(t-1)
31:                                   ENDIF
```

Fig. 9. Step 4: example of SRS (f_PDLCond and s_PDLCond) in definitional style.

An example of translating a specific timing function is given in Fig. 11. Lines 1–2 define the condition used in timing function `t_Trip`. The `cycletime` in line 3 is an interval between two consecutive executions.

2.2. PFS to PVS translation and PVS proof

Functional properties to be verified are derived from a natural language specification included in the PFS document. To provide systematic guidelines in the translation

```
 1: twf(C:pred[tick], t:tick, tv:tick): RECURSIVE bool =
 2:    IF t = 0 THEN FALSE                        % initial value is FALSE
 3:    ELSE EXISTS (ts: {t:tick | 0 < t}):
 4:         (t-tv+1) <= ts AND ts <= t AND        % During a time interval
 5:         (C(ts) AND NOT twf(ts-1))             % if it starts TRUE
 6:    ENDIF                                      %  with just before FALSE,
 7:  MEASURE t                                    % output is TRUE
 8:
 9: twfs(C:pred[tick], tv:tick) : pred[tick] =
10:     (LAMBDA (t:tick):twf(C,t,tv))
```

Fig. 10. Step 5(1): the semantics of timing functions.

```
1: C_Trip(t:tick) : bool = f_FaveC(t) >= k_FaveCPDL AND
2:                         (NOT t_Pending(t)) AND  s_Pending(t)
3: t_Trip(t:tick) : bool = twfs(C_Trip,k_trip/cycletime)(t)
```

Fig. 11. Step 5(2): translation of timing functions.

process, we propose a two-step process. In the first phase, cross-reference table containing relationship between the terms used in PFS and variables defined in SRS is used. See Table 1 for an example drawn from Wolsung SDS2 cross-references. It describes that low core differential pressure, commonly used and well-understood term among nuclear engineers, have been pointed to several different terms in the software requirements.

As it has been initially formulated, cross-reference table is too coarse to be useful in the software requirements verification process, and it is necessary to rely on domain experts to refine the table so that difference in the contexts becomes clearer. Table 2 shows an example of finer-grain cross-reference table used in the case-study. Such refinement is essential so that functional properties can be correctly verified within the right context.

In the second phase, one expresses functional properties to be verified using one of the following patterns. While functional properties are, by definition, application-dependent and therefore broad in scope and diversity, we found the following three patterns to be often needed when analyzing safety-critical real-time software requirements.

1. *Inputs-trigger-an-immediate-output*. This property proves that the occurrence of triggering event results in the generation of required output at the same time. For example, whenever *f_condition=k_condition* is satisfied, the output *f_output* is *k_output*. The conditions can be expressed by a few terms using 'AND' and 'OR'. When explicit timing constraints are missing, implicit universal quantifier is used so that property is satisfied at all times $t \geq 0$. In PVS theory, the first pattern is expressed as follows:

```
theorem_input_output: THEOREM
  (f_condition(t)=k_condition)=>
  (f_output(t)=k_output)
```

2. *Inputs-trigger-continuous-outputs*. While this pattern is similar to the first, difference is that the output must be maintained for specified time interval between t and (t+duration). Hypothetical situation includes continuous generation of alarms should the reactor pressure is found to exceed predefined threshold. In PVS, the property is captured as follows:

```
theorem_duration: THEOREM
  (f_condition(t)=k_condition)=>
  (FORALL (ti: tick):
  (t<=tiandti<=t+duration-1)=>
  (f_output(ti)=k_output))
```

3. *Input-trigger-a-delayed-output*. In some cases, one must pass a required duration (theorem_expiration1) or must continue to monitor environmental condition (theorem_expiration2) for a required duration before generating an output. Such properties are often enforced so that system may avoid generating unessential alarms in response to transient and temporary events.

```
theorem_expiration1: THEOREM
  (f_condition(t)=k_conditions)=>
  ((0<=duration)=>
  (f_output(t+duration)=k_output))
theorem_expiration2: THEOREM
  (f_initcondition(t)=k_initcondi-
  tion AN (FORALL (ti: tick):
  (t<=ti and ti<=t+duration-1)=>
  (f_staycondition(ti)=k_staycondi-
  tion))=>
  (f_output(t+duration)=k_output)
```

The translation from PFS to PVS THEOREMs follows the example in Fig. 12, which shows the translation of the items from Fig. 1. Item e.1 in Fig. 1 is 'If the D/I is open, select the *0.3%* full power (FP) conditioning level. If $\phi_{LOG} < 0.3\%$ *FP-50* mV, condition out the immediate trip. If $\phi_{LOG} \geq 0.3\%$ *FP*, enable the trip'. This sentence matches (Pattern 1), input–output property. 'The D/I' is described as 'hand switch' and 'low power conditioning level' in line 2 in Table 2. So 'the D/I' is mapped to 'm_PDLCond'. And 'the D/I is open' means that m_PDLCond(t)=k_CondSwLo. 'k_CondSwLo means open' is extracted from the constant list in the SRS documents. In this state, 'immediate trip' is 'condition out' when $\phi_{LOG} < 0.3\%$ *FP-50* mV. ϕ_{LOG} is mapped f_Flog (in Table 2) and *0.3% FP* is 2739 mV,

Table 1
A cross-reference in SRS (as given in the PFS document)

PFS	SRS
PHT low core differential pressure	f_PDLCond, f_PDLCondHA, f_PDLCondLA, f_PDLSnrDly[i],i=3,...,4, f_PDLSnrI[i],i=1,...,4, f_PDLDly, f_PDLSnrM[i],i=1,...,4, f_PDLTrip, t_Pending, t_Trip, f_PHTDAlm[i],i=1,...,4, f_PHTDErr, f_PHTDM, f_PDLSpI[i],i=1,...,4, f_SprdChkA

Table 2
Cross-references between terms in PFS and variables in SRS

PFS	SRS
Hand switch, low power conditioning level	m_PDLCond
Hand switch, pump operating mode	f_PumpMde
Previous conditioning status	s_PDLCond
PHT core differential pressure measurement, ΔP_i, DP signal	m_PHTD
PHT low core differential pressure parameter trip, ΔP_{trip}, parameter trip (D/O)	f_PDLTrip

that is, k_FlogPDLLo (extracted from the PFS and SRS). In this state, immediate trip should not operate (condition out). It can be written as f_PDLCond= k_CondOut. In this way, we translate THEOREM th_e_1_1. In a similar way, 'enable trip' when $\phi_{LOG} \geq 0.3\%$ *FP* is translated into THEOREM th_e_1_2.

A Pattern 2 shows THEOREM th_g_3_1_appropriate in Fig. 17. The THEOREM th_g_3_2 in Fig. 17 is a Pattern 3.

The translated specification is stored in a file for verification by PVS. The verification in PVS cannot be entirely automated, but we found that there is a pattern when we prove similar properties. The core parts of PVS specification are a list of function definitions which are translated through steps 4 and 5. When we prove theorems, we should rewrite and expand the theorems as these definitions. For example, the output of the THEOREM th_e_1_1 is f_PDLCond so we expand the definition of function f_PDLCond. The definition of f_PDLCond is removed and s_PDLCond, which is an input to f_PDLCond, is newly introduced by expanding the f_PDLCond. We can finish to prove the THEOREMs by expanding definitions. However, expanding definition chains have circular dependencies, the proof could be failed by falling into infinite rewriting. So we need more cautious proof steps. The 'expand' proof command in PVS expand function definitions, and 'grind' proof command do extensive expansion and rewriting definitions.

When we prove THEOREM th_e_1_1 and THEOREM th_e_1_2 in Fig. 12, f_PDLCond is a recursive definition. So we can prove them by (expand 'f_PDLCond')(grind:exclude ('f_PDLCond')).

A proof template is (expand* …) (grind:exclude (…)) or (grind:exclude (…)). The … is related to the functions or definitions on the paths of data-flows to avoid infinite rewriting.

3. Discussion

This section shows the usefulness of the verification of SCR-style SRS using PVS and some ambiguities which we found during our verification experience.

The general comparison between theorem proving and model checking was extensively studied, and we do not mention about the comparison results. However, we would like to express a few advantages of using PVS. One is the usage of 'undef.' PVS admit to specify an undefined value as undef keyword. An undefined value will be assigned a value during later phases of the software development process. It is difficult to manage the undefined value in model checking. Model checker can deal with explicit specified values, but a theorem prover such as PVS can manage value as a symbolic value without difficulty. Generally speaking, in early phase in software development, we cannot decide the values of some variables. So undef keyword is very useful to specify these values.

During our verification experience, we discovered notational errors, different terms for the same concepts, and hidden assumptions.

First, we found that different terms were used in PFS during the construction of the cross-references. If one term of PFS refers a few terms in SRS, readers of documents could be confused or misunderstand. For example, the m_PDLCond in the third line of Table 2, is used as hand switch, low power conditioning level, and conditioning level. The m_PHTD in the fourth line is used as core differential pressure measurement, ΔP_i, and DP signal. The f_PDLTrip, in the ninth line is used as the state of PHT low core differential pressure parameter trip, ΔP_{trip}, and parameter trip (D/O). Specifiers should replace these ambiguous terms with a single clear term. Our method can be therefore valuable in encouraging that the PFS use terms in the same way that the SRS does and can improve the quality of documents.

Second, other different terms in the PFS were 'condition out the immediate trip' and 'enable trip.' The 'condition out' is actually the opposite of 'enable', but this is far from clear. Our analysis highlights such obfuscated wording, in Fig. 13. We present a modified PFS term, e. 'the low power conditioning level' from 'the conditioning level' in Fig. 1. The 'condition in-enable' is also modified to 'disable–enable'.

Third, we found another ambiguity between PFS and SRS. The item g.0 in Fig. 1 introduces delayed trip conditions. The conditions for the delayed trip are 'any ΔP signal is either below the immediate trip setpoint (2610) or above the high irrational trip set point (80)'. And we translated the item e.0 into THEOREM th_g_0_1 in Fig. 14. This THEOREM was

```
th_e_1_1 : THEOREM (m_PDLCond(t) = k_CondSwLo AND f_Flog(t) < 2739-50)=>
           f_PDLCond(t) = k_CondOut
th_e_1_2 : THEOREM (m_PDLCond(t) = k_CondSwLo AND f_Flog(t) >= 2739) =>
           f_PDLCond(t) = k_CondIn
```

Fig. 12. Example of translation from PFS to PVS THEOREMs.

8: e. Determine the immediate trip conditioning status from *the lower power conditioning level*
9: D/I as follows:
10: 1. If the D/I is open, select the *0.3% FP* (Full Power) conditioning level.
11: If ϕ_{LOG} < *0.3%FP-50 mV*, *disable out the immediate trip*.
12: If ϕ_{LOG} >= *0.3%FP*, *enable the immediate trip*.

Fig. 13. Unambiguous PFS.

```
th_g_0_1: THEOREM FORALL (i: onefour):
     (m_PHTD(t)(i) < 2610 AND f_FaveC(t) > 80) =>
     f_PDLSnrDly(t)(i) = k_SnrTrip
th_g_0_2: THEOREM FORALL (i: onefour):
     NOT (m_PHTD(t)(i) < 2610 AND f_FaveC(t) > 80) =>
     f_PDLSnrDly(t)(i) = k_SnrNotTrip
```

Fig. 14. An ambiguity between PFS and SRS.

proved successfully. After this proof, we tried to check a complement condition, i.e. 'any ΔP signal is *neither* below the immediate trip setpoint *nor* above the high irrational trip set point'. This condition, shown in `THEOREM th_g_0_2` in Fig. 14, should not generate the (delayed) trip. However, we cannot prove the `THEOREM th_g_0_2`.

The unfinished proof status is shown in Fig. 15. The proof procedure is a procedure to make a proof tree, and we should prove the leaf nodes to finish the proof. In this proof, the root node of the proof tree is `THEOREM th_g_0_2` in Fig. 15, and the leaf nodes are `th_g_0_2.1` and `th_g_0_2.2` in Fig. 15. We cannot proceed to prove the leaf nodes any more, so we should investigate the cause.

The meaning of `th_g_0_2.1` is `{-1} m_PHTD(t!1)(i!1)<=2610 AND {-2} f_FaveC(t!1)=>80=>{1} f_FaveC(t!1)>80 OR {2} k_SnrNotTrip?(k_SnrTrip)`.

In this node, the `k_SnrNotTrip?(k_SnrTrip)` in `{2}` means `k_SnrNotTrip=k_SnrTrip`, so `{2}` is false.

`{-2} f_FaveC(t!1)=>80` and `{1} f_FaveC(t!1)>80` are inconsistency, i.e. if `f_FaveC(t!1)=80` is satisfied, the leaf node is not true. So the problem occurs at `f_FaveC(t)=80`.

In a similar way, `th_g_0_2.2` has a problem at `m_PHTD(t)(i)=2610`.

So we can conclude that the complement condition is not satisfied and it has an inconsistent delayed trip condition in SRS. We investigate the inconsistency between PFS and SRS, and then we conclude that it is not a violation of safety properties even though SRS does not confirm to PFS. The conditions, `m_PHTD(t)(i)=2610` and `f_FaveC(t)=80` are a kind of safety margin, that is, it is not necessary to generate the trip signal when this condition is satisfied but the system generate the trip signal. And, the trip signal does not make the system a dangerous state.

Fourth, there were hidden assumptions, such as in the following PFS. The g.1, g.2, and g.3 in Fig. 1 are translated into Fig. 16 in PVS. But we could not prove the `THEOREM th_g_3_1_inappropriate`.

If we fail to prove a theorem, there are three sources of incorrectness. One is the SRS, another is the PFS, and the third is proof procedure. Indeed, there is no general way to find what is wrong. It is a current limitation of theorem provers, however the unfinished proof status could be helpful to find the source of errors. Formal method researchers are continuing to improve ability to find errors as well as to prove critical properties [21]. We investigated the reason and we concluded that there were hidden assumptions. Items g.2 and g.3 in Fig. 1 are not independent. In other words, the item g.3 can be true only if the item g.2 is true. 'Once the delayed parameter trip has occurred' does not mean 'the delayed parameter trip has occurred' directly, but it means 'f_{AVEC} equals or exceeds *80% FP* and then the delayed parameter trip has occurred'. So the assumption *the delayed parameter trip has occurred* in item g.3 should be strengthened with items g.2.1 and g.2.2. As a result of this investigation, we translated the above PFS into PVS specifications again, such as in Fig. 17. Then we succeeded in the proof of `THEOREM th_g_3_1_agppropriate`. This error was not found through inspection, and is the kind of error that is difficult to find without formal analysis.

3.1. Related works

The most related research is done by Myla Archer in Naval Research Laboratory (NRL). NRL continues to work

```
th_g_0_2.1 :

{-1}  m_PHTD(t!1)(i!1) <= 2610
{-2}  f_FaveC(t!1) >= 80
  |-------
{1}   f_FaveC(t!1) > 80
{2}   k_SnrNotTrip?(k_SnrTrip)


th_g_0_2.2 :

{-1}  m_PHTD(t!1)(i!1) <= 2610
{-2}  f_FaveC(t!1) >= 80
  |-------
{1}   m_PHTD(t!1)(i!1) < 2610
{2}   k_SnrNotTrip?(k_SnrTrip)
```

Fig. 15. Unfinished proof of `THEOREM th_g_0_2`.

```
th_g_1 : THEOREM FORALL (t:tick):
       f_FaveC(t) >= k_FaveCPDL AND
       (EXISTS (i:onefour): m_PHD(t)(i) < k_PDLSpDly) AND
       s_PDLDly(t) = k_InDlyNorm
       =>
       (FORALL (t1: tick) :
        (t <= t1 AND t1 <=1000/cycletime+t) => f_PDLDly(t1) = k_InDlyNorm)
th_g_2_1 : THEOREM FORALL (t : tick) :
       s_Pending(t) = true AND t_Pending(t) = false AND
       f_FaveC(t) >= k_FaveCPDL AND t_Trip(t-1) = false
       => t_Trip(t)
th_g_2_2 : THEOREM FORALL (t: tick):
       s_Pending(t) = true AND t_Pending(t) = false AND
       f_FaveC(t) < k_FaveCPDL AND s_Trip(t) = false
       => t_Trip(t) = false
th_g_3_1_inappropriate : THEOREM FORALL (t:tick):
       s_Trip(t) = false AND t_Trip(t) = true =>
       FORALL(t1 : tick): ((t <= t1 and t1 <= t+1000/cycletime-1) =>
           t_Trip(t1) = true)
```

Fig. 16. Example of inappropriate translation of PFS.

on SCR, and the recent verification environment is timed automata modeling environment (TAME) [2]. They translated the SCR specification to PVS specification, and then they proved safety properties or invariants using PVS. It is quiet similar to our approach however their SCR model is based on event-action transition model, but our model is based on AND–OR table. It changes proof procedures. We should integrate two methods (or verification procedure) to verify more general system.

The work presented here is complemented by ongoing work at McMaster University by Lawford et al. [12]. Using a similar case-study, their work concentrates on verification of the refinement of the requirements in the SRS into design elements, also expressed in SCR, in the software design description (SDD). They use an extension of the four-variable model of Parnas [19] into a relational setting, and claim that their approach is more intuitive for system engineers. Our goal in the present work is essentially the same-to develop easier-to-use verification approaches-for application to the earlier part of the software development process.

Dutertre and Stavridou also provide an good case-study [4]. The model in the case-study is based on data-flow model as in our study and they also used PVS. However, the way to describing timing function is different from our approach, so the verification procedure is different. And we describe a basic function in a tabular notation, so we can check more internal consistencies. In addition, we studied declarative style as well as definitional style.

Another research from nuclear domain experts [22] shows that why our automated translation procedure is important for verifying safety-critical system. They wrote a SRS in colored petri nets (CPNs) and then manually translated the CPN model into PVS. We found some errors in the translated PVS specification. Those errors show that the automatic and systematic method is an important factor in a successful verification [11].

Another approach for formal validation of requirements from PFS is done by Gervasi and Nuseibeh [5]. It provides a systematic and automated method to construct a model from a PFS, and then checking some structural properties (for

```
th_g_3_1_appropriate : THEOREM FORALL (t:tick):
     s_Trip(t) = false AND t_Trip(t) = true AND
     %% strengthen assumption from th_appropriate_g_2_1~g_2_2
     f_FaveC(t)>=k_FaveCPDL AND t_Pending(t)=false AND s_Pending(t)=true =>
     FORALL(t1 : time): ( (t <= t1 and t1 <= t+1000/cycletime-1) =>
        t_Trip(t1) = true)
th_g_3_2 : THEOREM FORALL (t:tick):
      ( s_Trip(t) = false AND t_Trip(t) = true
        f_FaveC(t) >= 80 AND t_Pending(t)=false AND s_Pending(t)=true AND
        ( FORALL(t1 : time): ( (t <= t1 and t1 <= t+1000/cycletime-1) =>
          t_Trip(t1) = true) )
      => (FORALL (i:onefour): m_PHD(t)(i) > k_PDLSpDly) OR
         (f_FaveC(t) < k_FaveCPDL)
         => t_Trip(t) = false
```

Fig. 17. Example of appropriate translation.

example, function's domain is correct) of the constructed model. We think that their extraction technique can help in extracting functional properties; however, they do not verify extracted functional properties.

4. Conclusion

To verify functional properties, we developed a software tool with a graphical user interface that converts SCR-style requirements specifications into the PVS language. In addition, we provide a method for verifying functional properties in PFS using PVS. We believe that the procedure helps to construct a high-quality safety-critical software.

Our graphical editor provides a user-friendly interface to allow editing of SCR-style specifications and automates the translation process. However, the proof process can be completed with a limited study of the proof pattern. The specifier translates PFS into PVS theorems manually, even though we can translate systematically using a cross-reference table.

Although we strongly believe that our approach delivers significant benefits to practitioners, the following further enhancements seem to be desirable:

- integration with an environment for verifying structural properties which was previously developed [10];
- development of translation rules so that a formal specification written in statecharts or modecharts can be verified using the same approach;
- more systematic method of translating from PFS to PVS theorems, to enhance completeness of the current cross-reference methods;
- additional study of proof patterns, to the verification;
- enhancements to the SRS-style editor, such as XML translation, to increase its practical utility.

Acknowledgements

This work was partially supported by the Korea Science and Engineering Foundation through the Advanced Information Technology Research Center and by the National Science Foundation under grants CCR-00-82560 and CCR-00-86096.

References

[1] AECL CANDU. Software work practice. Procedure for the specification of software requirements for safety critical systems, Wolsung NPP, 00-68000-SWP-002 1991.
[2] Archer M. TAME: using PVS strategies for special-purpose theorem proving. Ann Math Artif Intell 2000;29:139–81.
[3] Crow J, Owre S, Rushby J, Shankar N, Srivas M. A tutorial introduction to PVS Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95) 1995 p. 1–112.
[4] Dutertre B, Stavridou V. Formal requirements analysis of an avionics control system. IEEE Transact Software Eng 1997;23(5):267–78.
[5] Gervasi V, Nuseibeh B. Lightweight validation of natural language requirements: a case study. Proceedings of 4th IEEE International Conference on Requirements Engineering (ICRE 2000) 2000.
[6] Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language LUSTRE. Proc IEEE 1991;79(9): 1305–20.
[7] Heitmeyer C, Kirby J, Labaw B. The SCR method for formally specifying, verifying and validating software requirements: tool support. Proceedings of the 19th International Conference on Software Engineering (ICSE '97) 1997 p. 610–1.
[8] Heimdahl MPE, Leveson NG. Completeness and consistency in hierarchical state-based requirements. IEEE Transact Software Eng 1996;22(6):363–77.
[9] Hinchey M, Bowen J. Application of formal methods. Englewood cliffs, NJ: Prentice-Hall; 1995 p. 1–442.
[10] Kim T, Cha S. Automated structural analysis of SCR-style software requirements specifications using PVS. J Software, Testing, Verification, Reliab 2001;11(3):143–63.
[11] Kim T, Cha S. Comment on: development of a safety critical software requirements verification method with combined CPN and PVS: a nuclear power plant protection system application. Reliab Eng Syst Safety 2004;83(1):121–2.
[12] Lawford M, McDougall J, Froebel P, Moum G. Practical application of functional and relational methods for the specification and verification of safety critical software. Proceedings of Algebraic Methodology and Software Technology, the 8th International Conference (AMAST 2000), LNCS 1816 2000 p. 73–88.
[13] Lutz RR. Targeting safety-related errors during software requirements analysis. Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering 1993;99–106.
[14] Miller SP, Srivas M. Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. Workshop on Industrial-Strength Formal Specification Techniques (WIFT' 95) 1995 p. 2–16.
[15] Owre S, Shankar N, Rushby J, Stringer-Calvert D. PVS system guide version 2.4.: Computer Science Laboratory, SRI International; 2001 p. 1–97.
[16] Owre S, Shankar N, Rushby J, Stringer-Calvert D. PVS language reference version 2.4.: Computer Science Laboratory, SRI International; 2001 p. 1–102.
[17] Owre S, Rushby J, Shankar N, von Henke F. Formal verification for fault-tolerant architecture: prolegomena to the design of PVS. IEEE Transact Software Eng 1995;21(2):107–25.
[18] Owre S, Rushby J, Shankar N. Integration in PVS: tables, types, and model checking. Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97) 1997 p. 366–83.
[19] Parnas D, Madey J. Functional documentation for computer systems engineering. Technical Report CRL No. 273, Telecommunications Research Institute of Ontario, McMaster University; 1991.
[20] Rushby J, Srivas M. Using PVS to prove some theorems of David Parnas Proceedings of International Conference on HOL Theorem Proving and Its Applications 1993 p. 163–73.
[21] Rushby J. From refutation to verification Proceedings of Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE XIII/PSTV XX 2000) 2000 p. 369–74.
[22] Son H, Seong P. Development of a safety critical software requirements verification method with combined CPN and PVS: a nuclear power plant protection system application. Reliab Eng Syst Safety 2003;80(1):19–32.
[23] Shankar N, Owre S, Rushby J, Stringer-Calvert D. PVS prover guide version 2.4.: Computer Science Laboratory, SRI International; 2001 p. 1–127.