Formal Verification of Functional Properties of an SCR-Style Software Requirements Specification Using PVS*

Taeho Kim[1,2], David Stringer-Calvert[2], and Sungdeok Cha[1]

[1] Department of Electrical Engineering and Computer Science and
Advanced Information Technology Research Center (AITrc),
Korea Advanced Institute of Science and Technology (KAIST)
Taejon 305-701, Korea
{thkim, cha}@salmosa.kaist.ac.kr
[2] Computer Science Laboratory, SRI International
Menlo Park CA 94025, USA
dave_sc@csl.sri.com

Abstract. Industrial software companies developing safety-critical systems are required to use rigorous safety analysis techniques to demonstrate compliance to regulatory bodies. While analysis techniques based on manual inspection have been successfully applied to many industrial applications, we demonstrate that inspection has limitations in locating complex errors in software requirements.

In this paper, we describe the formal verification of a shutdown system for a nuclear power plant that is currently operational in Korea. The shutdown system is an embedded real-time safety-critical software, and has a description in a Software Cost Reduction (SCR) style specification language. The key component of the work described here is an automatic method for translating SCR-style Software Requirements Specifications (SRS) into the language of the PVS specification and verification system. A further component is the use of property templates to translate natural language Program Functional Specifications (PFS) into PVS, allowing for high-assurance consistency checking between the translated SRS and PFS, thereby verifying the required functional properties.

1 Introduction

Various approaches have been suggested for developing high-quality requirements specifications and conducting cost-effective analysis. Although inspection [1] can, in principle, detect all types of errors in requirements, experience in conducting inspections on the Software Requirements Specification (SRS) for

* This work was 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.

J.-P. Katoen and P. Stevens (Eds.): TACAS 2002, LNCS 2280, pp. 205–220, 2002.

the Wolsung[1] shutdown system number 2 (SDS2) revealed that inspection has potentially lethal limitations in demonstrating safety.

The Wolsung SDS2 is designed to continuously monitor the reactor state (e.g., temperature, pressure, and power) and to generate a trip signal (e.g., shutdown command, and display) if the monitored variables exceed predetermined safety parameters. The SDS2 SRS specifies 30 monitored variables (inputs from the environment), 59 controlled variables (outputs to the environment), and 129 computational functions relating them. The SRS is 374 pages in length and was subject to four relatively minor releases in less than a year. Inspection of the initial release of the SRS, conducted by four staff members, to validate consistency between the SRS and the natural language Program Functional Specification (PFS) took about 80 staff hours of formal inspection meetings, during which only 17 trivial notational errors and incomplete definitions in the PFS and SRS were discovered.

This experience with manual inspection motivated research to explore more robust and rigorous methods of analysis. To this end, (1) we provide an automatic method for translating SRS into the language of the PVS specification and verification system [2], and we implemented a tool for editing and translating, and (2) we translate from PFS into PVS using property templates and cross reference. Last, (3) we verify the consistency between translated SRS and PFS. In this case study, we concentrate on one trip condition (PDL trip), among three trip conditions, for which SRS is 22 pages, and PFS is 4 pages. The whole SRS for SDS2 is 374 pages, and the whole PFS is 21 pages.

Even though our example case study is in the nuclear domain, we believe the verification procedures we propose are general and applicable to wide range of safety-critical systems.

The rest of our paper is organized as follows. In Section 2, we review how an SCR-style software requirements specification, which is used in Wolsung SDS2, is organized. Section 3 describes the verification procedure developed, detailing the case study of the Wolsung SDS2, and Section 4 discusses results and comparisons with other approaches. Finally, Section 5 concludes this paper.

2 Background

2.1 SCR-Style SRS

An SCR-style formal specification [3] has four key attributes:

- Variable definitions
- Functional overview diagrams (FODs)
- Structured decision tables (SDTs)
- Timing functions

[1] The Wolsung nuclear power plant in Korea, used as a case study in this paper, is equipped with a software-implement emergency shutdown system.

It is slightly different from the SCR specification language developed by researchers at the Naval Research Laboratory and supported by the SCR* toolset [4]. The difference lies in how primitive functions are described - where SCR style uses a time-triggered AND-OR table, the SCR* uses an event-action table format. A system written in SCR-style requirements is designed to read monitor variables for an external environment (e.g., temperature, pressure, and power) and to generate control values (e.g., a shutdown command).

The detailed description of the attributes of an SCR-style SRS as follows:

Variable definitions : The interface between the computer system and its environment is described in terms of monitored and controlled variables. Monitored variables, whose names start with the `m_` prefix, refer to the inputs to the computer system, and controlled variables, whose names start with the `c_` prefix, refer to the outputs from the computer system. A variable may be analog or digital.

Functional Overview Diagrams (FODs) : An FOD illustrates, in a notation similar to data flow diagrams, a hierarchical organization of functions. A group, denoted by the `g_` prefix, consists of subgroups or basic functions. Each basic function name starts with the `f_` prefix. For example, the group `g_Overview`, illustrated in figure 1.(a), is refined into `g_ProcessInputs`, `g_PDL`, `g_PZL`, `g_SLL` groups as shown in figure 1.(b). The `g_ProcessInputs` is a preprocessor for the system. `g_PDL`, `g_PZL`, and `g_SLL` are trip signals for returning the system to a safe state.

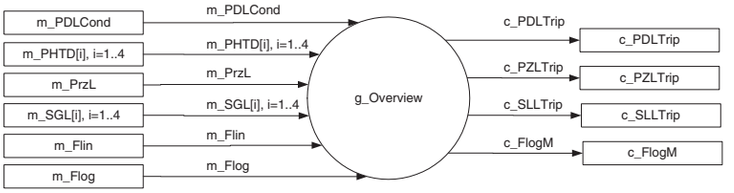
Similarly, the group `g_PDL` is composed of six basic functions and two timing functions as shown in figure 1.(c). A basic function is a mathematical function with zero delay and are specified in a structured decision table. Outputs are synchronous with inputs in a basic function. The `s_` prefix denotes a state name, used to store the previous value of a function, that is, with one clock delay. Timing functions are drawn as a bar (|), for example, `t_Pending` and `t_Trip` in figure 1.(c).

In addition to the hierarchical relations, the FOD specifies inputs, outputs, and internal data dependencies among various components. Such data dependencies implicitly dictate the proper order of carrying out a set of functions. For example, in figure 1.(c), the output of the `f_PDLSnrI` function is used as an input to the `f_PDLTrip` function, and the latter function therefore may be invoked only when the former is completed. This is the same concept used in dataflow languages such as LUSTRE [5].

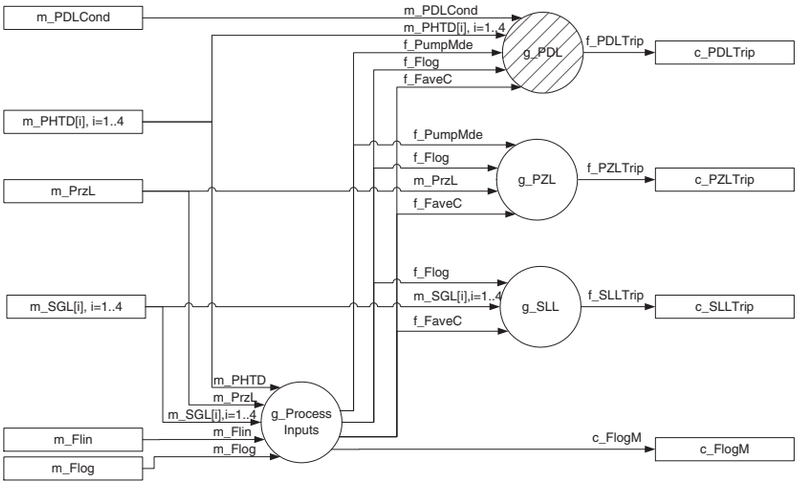
Structured Decision Table (SDT) : The required behavior of each basic function is expressed in a tabular notation, called SDT, as shown in figure 2. The function `f_PDLCond` produces an output, whose value is either `k_CondOut` or `k_CondIn`. The `k_` prefix indicates a constant value.

Condition macros are a substitution for specific conditions. For example, lines 2–5 of the condition macros in figure 2 define the macro `w_FlogPDLCondLo [f_Flog]`. If `f_Flog<k_FlogPDLLo-k_CondHys`, `w_FlogPDLCondLo[f_Flog]` is denoted "`a`" according to line 3.

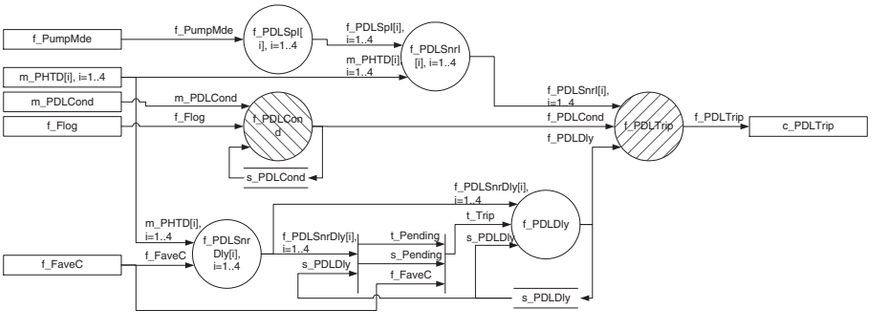
As shown in the second column in the SDT, this function returns the value `k_CondOut` when the value `m_PDLCond` is equal to `k_CondSwLo` and



(a) A part of the FOD for SDS2



(b) A lower-level FOD of `g_Overview`



(c) A lower-level FOD of `g_PDL`

Fig. 1. Examples of the function overview diagram

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
6: w_FlogPDLCondHi[f_Flog]
7: a f_Flog < k_FlogPDLHi - k_CondHys
8: b f_FlogPDLHi - k_CondHys <= f_Flog < k_FlogPDLHi
9: c f_Flog >= k_FlogPDLHi
Structured Decision Table:

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

Fig. 2. The SDT for f_PDLCond

`w_FlogPDLCondLo[f_Flog]` is equal to `a`. The '-' entries denote the 'don't care' condition.

Timing function : Timing functions are used for specifying timing constraints and real-time behavior. A prototype of a timing function is `t_Wait` is `t_Wait( C(t), Time_value, tol)`, where *`C`*`(t)` is a logical condition at time `t`, the *`Time_value`* is a time interval, and *`tol`* is an acceptable time deviation. Intuitively speaking, the function stays true during *`Time_value`* when the immediately previous value of the function is false and *`C`*`(t)` is true at time `t`. The `t_Wait` at time `0` is `FALSE`. The formal semantic definition of a timing function is

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

For example, `t_Trip` in figure 1.(c) is defined such that

```
t_Trip = t_Wait(C, k_PDLTrip, k_PDLTripTol)
   where C = (f_FaveC >= k_FaveCPDL AND
                t_Pending = false AND s_Pending = true)
```

This means that `t_Trip` is true between time `t` and time `t + k_PDLTrip` when `t_Trip` is false at time `t`-ϵ, and `f_FaveC >= k_FaveCPDL AND t_Pending = false AND s_Pending = true`. The `k_PDLTripTol` is the tolerance of
`k_PDLTrip`.

2.2 Program Functional Specification (PFS)

A program functional specification (PFS) is a system specification written in natural language (English for Wolsung SDS2), as prepared by domain experts. The structure is highly intuitive, and an example is shown in figure 3. The PFS for SDS2 is 21 pages, and PDL trip in this case study accounts for 4 pages.

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 $\phi_{LOG} < 0.3\%FP - 50mV$, condition out the immediate trip.
12: If $\phi_{LOG} >= 0.3\%FP$, enable the trip.
13: · · · · · ·
14: g. If no PHT ΔP delayed trip is pending or active then execute a delayed
15: trip as follows:
16: 1. Continue normal operation without opening the parameter trip D/O for
17: nominally three seconds.
18: 2. After the delay period has expired, open the parameter trip D/O
19: if f_{AVEC} equals or exceeds $80\%\overline{FP}$.
20: Do not open the parameter trip D/O if f_{AVEC} is below $80\%FP$.
21: 3. Once the delayed parameter trip has occurred,
22: keep the parameter trip D/O open for one second.
23: · · · · · ·
24: h. Immediate trips and delayed trips (pending and active) can occur simultaneously.
25: · · · · · ·

Fig. 3. Example of program functional specification

3 Verification of SCR-Style SRS

3.1 Translation from SCR-Style SRS to PVS

We describe a translation procedure of SCR-style SRS as embodied in our tool, and its application to the specific case study of the Wolsung SDS2 SRS. The translation procedure consists of 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
5. Definition and translation of timing functions

Step 1. Definition of Time Model Elements:
Time increases by a fixed period, so time can be specified using a `tick`, a positive number. A time is represented by the set of successive multiples of that period, starting from 0. This part is common through different specifications and is denoted in figure 4.[2] Time is described in the type `tick` definition in line 1, being declared as a `nat` (natural number). Line 2 defines `t`, representing a variable of type `tick`. In line 3, a constant `init` is defined to be `0`, for use as the initial value of `tick`.

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

Fig. 4. Step 1. Definition of model elements

Step 2. Definition of Types and Constants:
The type of a variable in SCR-style SRS is different for analog variables and digital variables. The type for an analog variable is declared to be a real number (or subtypes of real), and the type for a digital variable is a given enumeration. Trajectories of the value of variables with time are declared as functions from `tick` to the variable type.

Figure 5 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` will be used for constants whose values are undefined. An undefined value will be assigned a value during later phases of the software development process. `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 the trivial function mapping from any `tick` value `t` to the constant `k_Trip`. `to_CondInOut` is a 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] 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.

```
 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 var.
 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}      % incl. t->k_Trip
15: t_CondInOut : TYPE = [tick -> to_CondInOut]
16:
17: enumabc : TYPE = {a,b,c}
```

Fig. 5. Step 2. Definition of types and constants

Step 3. Definition of Types for Monitored and Controlled Variables: 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`. Figure 6 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`.

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

Fig. 6. Step 3. Definition of types for monitored and controlled variables

Step 4. Translation of SDTs:
Functions in an SCR-style SRS are structured in a hierarchy. The lowest level of the hierarchy is an internal computation function expressed as an SDT or a timing function. The hierarchical information is not needed in the translation for checking functional correctness; hence, this step translates only the SDT and timing functions.

There are two kinds of function. One is a function that reads values at tick t and writes values at tick t. The other is a function which reads both values at tick t and at t-1 and writes values at tick t. SCR-style SRS assumes that it takes zero time to execute a function.

Let *f_output, f_input1, f_input2*, and *s_output* be function names or variable names. The first kind of function is

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

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:tick):value_type = compute(f_input1(t),f_input2(t))
```

If the condition macro is defined within `compute`, the macro should be locally defined by the `LET` $\cdots$ `IN` construct. In this case, the translation template is[3]

```
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

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

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

In the second kind of a function, there is a circular dependency among the *f_output* and the *s_output*. The type checking of PVS does not admit circular dependencies in an explicit manner, so we use a definitional style with local definitions embedded within a recursive function, in this paper. The translation template for this kind of function introduces a local copy of the mutually dependent function.

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

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*

[3] In SCR-style SRS, functions and condition macros are defined as tabular notation, so `condition_macro` and `compute`s in translated PVS specification are expressed as a `TABLE` $\cdots$ `ENDTABLE` construct.

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 translation of f_PDLCond in figure 2 is shown in figure 7. f_PDLCond at line 4 is recursively defined, so we define f_PDLCond as a recursive function using RECURSIVE. And we define condition macro w_FlogPDLCond and w_FlogPDLCondHi in lines 6–11.

We also explored an approach using AXIOMs to introduce mutually recursive functions. The approach separates the definition part and declaration part in a way similar to high-level languages, so it does not need local definition. However, a step-by-step proof may be required for safety auditing, so there is a tradeoff between automation and auditability. We chose to prefer automation, as an aid to finding errors quickly, rather than fully auditable verification.

The translated specification in this paper is more complex than the declarative style because of the local definition and recursive definition for circular dependent functions. The major advantage of the definitional style is that it enables greater automation of proofs. However, the step-by-step proof that may be required for safety auditing is sometimes difficult. The declarative style supports less automation for proving, but allows for auditing the proof. We recommend the declarative style for early prototyping and the definitional style for full specifications.

Step 5. Definition and Translation of Timing Functions:
The semantics of timing functions in SCR-style SRS is given in figure 8. 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 sequence of the function twf

An example of translating a specific timing function is given in figure 9. Lines 1–2 define the condition used in timing function t_Trip. cycletime in line 3 is an interval between two consecutive executions.

3.2 Translation from PFS to PVS

The Program Functional Specification (PFS) is translated into PVS to check consistency between the PFS and the SRS. In this paper, we extract properties to be checked from the PFS, but generally they are not limited to those from the PFS. FMEA (Failure Mode and Effects Analysis) results and domain experts' knowledge also could be used to generate putative theorems that may be proven of the system under analysis.

The PFS is written in unconstrained natural language, so the translation cannot be easily automated. However, we propose a systematic two-phase process—the first phase is to define a cross-reference between terms in PFS and SRS. The second phase is to translate sentences in PFS into PVS. During the first phase, we can often find inconsistent terms, that must be resolved by the original specification authors. The second phase also cannot be automated, but there

```
 1:  f_PDLCond(t:tick) : RECURSIVE to_CondInOut =
 2:    LET
 3:     s_PDLCond : t_CondInOut =  LAMBDA (tt:tick):IF tt=0 THEN k_CondIn
 4:                                ELSE f_PDLCond(tt-1)
 5:                                ENDIF,
 6:     w_FlogPDLCondLo : enumabc  =  TABLE
 7:        ...                        % similar to if-then-else
 8:        ENDTABLE,
 9:     w_FlogPDLCondHi :enumabc =  TABLE
10:        ...                        % similar to if-then-else
11:        ENDTABLE,
12:      X = (LAMBDA (x1: pred[bool]),
13:                  (x2: pred[enumabc]),
14:                  (x3: pred[enumabc]),
15:                  (x4: pred[bool]) :
16:         x1(m_PDLCond(t) = k_CondSwLo) &
17:         x2(     w_FlogPDLCondLo) &
18:         x3(          w_FlogPDLCondHi) &
19:         x4(               s_PDLCond(t) = k_CondOut))  IN TABLE
20:        %    |    |    |    |
21:        %    v    v    v    v
22:        %-------|----|----|----|------------%
23:        | X( T  , a? , dc , ~ )| k_CondOut ||
24:        %-------|----|----|----|------------%
25:        | X( T  , b? , dc , T )| k_CondOut ||
26:        %-------|----|----|----|------------%
27:        | X( T  , b? , dc , F )| k_CondIn  ||
28:        %-------|----|----|----|------------%
29:        | X( T  , c? , dc , ~ )| k_CondIn  ||
30:        %-------|----|----|----|------------%
31:        | X( F  , dc , a? , ~ )| k_CondOut ||
32:        %-------|----|----|----|------------%
33:        | X( F  , dc , b? , T )| k_CondOut ||
34:        %-------|----|----|----|------------%
35:        | X( F  , dc , b? , F )| k_CondIn  ||
36:        %-------|----|----|----|------------%
37:        | X( F  , dc , c? , ~ )| k_CondIn  ||
38:        %-------|----|----|----|------------%
39:        ENDTABLE
40:  MEASURE t
41:
42:  s_PDLCond(t:tick):to_CondInOut =  IF t = 0 THEN k_CondIn
43:                                    ELSE f_PDLCond(t-1)
44:                                    ENDIF
```

Fig. 7. Example of definitional style of SRS (f_PDLCond and s_PDLCond)

```
 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. 8. 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. 9. Step 5 (2). Translation of timing function

are three distinct classes, or 'patterns,' in the text of the PFS. Because of the real-time constraints involved, these patterns cannot be described in temporal logic classes such as LTL (Linear Temporal Logic) or CTL (Computational Tree Logic), so we directly encode in a classical logic. Many researches have proposed real-time extension of temporal logics, but there is no standard notation for this. **(Pattern 1) Input-Output specifications** are requirements relating the input and output of functions. If *f_condition*(t) = *k_condition* at tick t, the output *f_output* is *k_output*. They can be described as an implication (with implicit universal quantification over tick t) as a relation:

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

(Pattern 2) Time-Duration specifications are real-time requirements such that if certain inputs are satisfied, the certain outputs should be maintained for a specified duration. If *f_condition*(t) = *k_condition* at tick t, the output of the *f_output* is *k_output* between tick t and t + *duration*.

```
theorem_duration : THEOREM FORALL (t:{ts:tick|ts>0}) :
    (f_condition(t) = k_condition) =>
        (FORALL (ti: tick): (t <= ti and ti <= t+duration) =>
            f_output(ti) = k_output)
```

(Pattern 3) Time-Expiration specifications are real-time requirements such that if certain inputs are satisfied and a specified duration has elapsed, then a certain output should be generated. If *f_condition*(t) = *k_condition* at tick t and tick *duration* has elapsed, the output of the *f_output* is changed to *k_output*.

```
theorem_expiration : THEOREM FORALL (t:{ts:tick|ts>0}) :
    (f_condition(t) = k_condition) =>
      (( 0 <= duration) => f_output(t+duration+1) = k_output)
```

The translation from PFS to PVS `THEOREM`s follows the example in figure 10, which shows the translation of the items from figure 3. Item e.1 in figure 3 is 'If the D/I is open, select the $0.3\%FP$ (Full Power) conditioning level. If $\phi_{LOG} < 0.3\%FP - 50mV$, condition out the immediate trip. If $\phi_{LOG} >= 0.3\%FP$, enable the trip.' This sentence matches (Pattern 1), input-output specifications. 'The D/I' is described as 'hand switch' and 'low power conditioning level' in lines 3 and 4 in figure 3. So 'the D/I' is mapped to '`m_PDLCond`.' And 'the D/I is open' means that `m_PDLCond(t) = k_CondSwLo`. In this state, 'immediate trip' is 'condition out' when $\phi_{LOG} < 0.3\%FP - 50mV$. ϕ_{LOG} is mapped `f_Flog` and $0.3\%FP$ is 2739 mv, that is, `k_FlogPDLLo`. This information is described in an appendix of PFS and SRS. In this state, immediate trip should not operate (condition out). It can be written as `f_PDLCond = k_CondOut`. In a similar way, 'enable trip' when $\phi_{LOG} >= 0.3\%FP$ translates `THEOREM th_e_1_2`.

```
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. 10. Example of translation from PFS to PVS THEOREMs

3.3 Verification

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. A proof template is `(expand* "...")(grind :exclude ("..."))` or `(grind :exclude ("..."))`. The ... is related to the functions or definitions on the paths of dataflows. The PVS proof strategy `grind` tries to rewrite the definitions in all possible cases, and for circular definition it rewrites infinitely. So ... in `exclude` are definitions are circular dependency relations. `expand` is used for rewriting only one expansion of a definition. When we prove `THEOREM th_e_1_1` and `THEOREM th_e_1_2` in figure 3.2, `f_PDLCond` is a recursive definition. So we can prove them by `(expand "f_PDLCond") (grind :exclude ("f_PDLCond"))`.

4 Discussion

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. For example, the m_PDLCond is used as hand switch, low power conditioning level, and conditioning level. The m_PHTD is used as Core Differential Pressure measurement, ΔP_i, and DP signal. The f_PDLTrip, is used as the state of PHT low core differential pressure parameter trip, ΔP_{trip}, and parameter trip(D/O). Our method can be therefore valuable in encouraging that the PFS use terms in the same way that the SRS does.

Second, other different terms in the PFS are '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 figure 11. We present a modified PFS term, e.'the low power conditioning level' from 'the conditioning level' in figure 3. The 'condition in - enable' is also modified to 'disable - enable'.

e. Determine the immediate trip conditioning status from *the low power conditioning level* D/I as follows:

1. If the D/I is open, select the $0.3\%FP$ conditioning level. If $\phi_{LOG} < 0.3\%FP - 50mV$, *disable the immediate trip.* If $\phi_{LOG} >= 0.3\%FP$, *enable the immediate trip.*

Fig. 11. Unambiguous PFS

Third, there are hidden assumptions, such as in the following PFS. The g.2 and g.3 in figure 3 are translated into figure 12 in PVS. But we could not prove the `THEOREM th_inappropriate_g_3`.

```
th_appropriate_g_2_1 : THEOREM FORALL (t:{ts:tick|ts>0}) :
        f_FaveC(t)>= 80 AND t_Pending(t) = false AND
        s_Pending(t) = true AND t_Trip(t-1) = false
        => t_Trip(t)
th_appropriate_g_2_2 : THEOREM FORALL (t:{ts:tick|ts>0}) :
        f_FaveC(t)< 80 AND t_Pending(t) = false AND
        s_Pending(t) = true AND t_Trip(t-1) = false
        => t_Trip(t)
th_inappropriate_g_3 : THEOREM FORALL (t:{ts:tick|ts>0}):
        t_Trip(t-1) = false AND t_Trip(t) = true =>
        FORALL(t1 : tick): ((t <= t1 and t1 <= 1000/cycletime +t) =>
             t_Trip(t1) = true)
```

Fig. 12. Example of inappropriate translation of PFS

We investigated the reason and we concluded that there were hidden assumptions. Items g.2 and g.3 in figure 3 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 figure 13. Then we succeeded in the proof of `THEOREM th_appropriate_g_3`. This error was not found through inspection, and is the kind of error that is difficult to find without formal analysis.

```
th_appropriate_g_3 : THEOREM FORALL (t:{ts:time|ts>0}):
    t_Trip(t-1) = false AND t_Trip(t) = true AND
    %% strengthen assumption from th_appropriate_g_2_1~g_2_2
    f_FaveC(t) >= 80 AND t_Pending(t)=false AND s_Pending(t)=true =>
    FORALL(t1 : time): ( (t <= t1 and t1 <= 1000/cycletime + t) =>
        t_Trip(t1) = true)
```

Fig. 13. Example of appropriate translation

Related Work

The work presented here is complemented by ongoing work at McMaster University by Lawford et al. [6]. 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 4-variable model of Parnas [7] 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.

Another approach for formal validation of requirements from PFS is done by Gervasi and Nuseibeh [8]. It provides a systematic and automated method to construct a model from a PFS, and then checking some structural properties (for 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 check functional properties.

5 Conclusion

Based on our experience of inspecting the Wolsung SDS2 SRS, we have demonstrated that inspection has limitations. 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 high-quality safety-critical software.

Users of our approach need not be experts on formal methods or power users of PVS. 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:

- 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

References

1. M. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, 12(7), pp. 133-144, 1986.
2. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, *PVS System Guide, PVS Language Reference, and PVS Prover Guide Version 2.4* , Computer Science Laboratory, SRI International, 2001.
3. AECL CANDU, *Software Work Practice, Procedure for the Specification of Software Requirements for Safety Critical Systems*, Wolsung NPP, 00-68000-SWP-002, 1991.
4. C. Heitmeyer, J. Kirby, and B. Labaw, "The SCR Method for Formally Specifying, Verifying and Validating Software Requirements: Tool Support," *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pp. 610-611, 1997.
5. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, 79(9), pp. 1305-1320, 1991.
6. M. Lawford, J. McDougall, P. Froebel, and G. Moum, "Practical application of functional and relational methods for the specification and verification of safety critical software," *Proceedings of Algebraic Methodology and Software Technology, 8th International Conference (AMAST 2000)*, LNCS 1816, pp. 73-88, 2000.
7. D. Parnas and J. Madey, "Functional documentation for computer systems engineering," Technical Report CRL No. 273, Telecommunications Research Institute of Ontario, McMaster University, 1991.
8. V. Gervasi and B. Nuseibeh, "Lightweight Validation of Natural Language Requirements: a case study," *Proceedings of 4th IEEE International Conference on Requirements Engineering (ICRE 2000)*, 2000.