

PVS를 이용한 SCR 스타일의 소프트웨어 요구사항 명세에서 기능 요구 사항의 정형 검증

(Formal Verification of Functional Properties of an
SCR-style Software Requirements Specifications using PVS)

김 태 호 ^{*} 차 성 덕 ^{††}

(Taeho Kim) (Sungdeok Cha)

요 약 소프트웨어의 개발 단계 중 품질을 결정하는 주요 단계는 요구 명세 단계로 알려져 있다. 따라서, 소프트웨어 개발 업체는 소프트웨어 요구명세의 분석을 가장 중요한 단계 중 하나로 취급하고 있고, 특히 안전성이 중요한 시스템의 경우에는 시스템을 운영하기 위하여 국내와 국제적인 규제 기관에서는 요구 명세의 분석을 통한 안전성의 입증을 요구한다. 소프트웨어의 요구 명세 분석을 위한 방법 중 인스펙션과 정형 검증이 가장 효과적인 방법으로 알려져 있다.

본 논문에서는 SCR-style의 요구 명세를 정리 증명기인 PVS를 이용하여 정형 검증을 수행하는 방법을 제안하였다. 그리고, 논문에서 제안된 방법으로 실제 월성 원자력 발전소의 정지 시스템의 검증을 수행하였다. 이 시스템은 인스펙션으로 검증된 적은 있으나 정형 검증 방법으로는 증명된 적이 없고, 국내에서 실제 운영되는 산업계시스템에 정형 검증 방법이 적용된 사례는 매우 드물기 때문에 차후 정형 검증 방법을 적용하기 위한 평가로서도 이와같은 실험적인 적용이 매우 중요하다.

키워드 : 정형 명세, 정형 검증, 정형 방법론, 정리 증명기

Abstract Among the many phases involved in software development, requirements analysis phase is generally considered to play a crucial role in determining the overall software quality. Therefore, many software development companies manage the phase as one of the important phase. Especially, safety assurance through requirements analysis for safety-critical systems is quite demanding, and national and international bodies routinely require safety demonstration. Among various approaches, inspection and formal methods are generally shown to be effective.

In this paper, we propose a formal verification procedure for SCR(Software Cost Reduction)-style SRS (Software Requirements Specification) using the PVS specification and verification procedure and applied this procedure to an industrial system such that a shutdown system for Wolsung nuclear power plant. This system had been verified through inspection not formal verification. The application of formal methods is rare in Korea, so it is very important to experiment about formal verification to industrial systems.

Key words : formal specification, formal verification, formal methods, theorem proving

1. 서 론

소프트웨어의 개발 단계 중 품질에 영향을 미치는 가장 중요한 단계는 소프트웨어의 요구 분석 단계로 알려져 있다. 즉, 오류가 개발 단계의 후기에 발견된다면 오

· 본 연구는 청단정보기술 연구센터를 통하여 과학재단의 지원을 받았음
^{*} 비 회 원 : 한국과학기술원 전기전산학과

thkim@salmosa.kaist.ac.kr
^{††} 총신회원 한국과학기술원 전자전산학과 교수
cha@salmosa.kaist.ac.kr

논문 접수 : 2000년 8월 2일
심사 완료 : 2001년 11월 9일

류를 수정하는데 개발 초기에 수정하는 것보다 비용과 노력이 많이 필요하다. 또한, NASA의 자료에 의하면 NASA에 의해 개발되어 운영되는 소프트웨어에서 발견되는 오류의 약 75%가 소프트웨어 요구 명세 단계에서 발생한 오류가 원인이 된다고 한다[1].

따라서, 소프트웨어 요구 명세(SRS; Software Requirements Specification)의 분석을 소프트웨어 개발 업체는 개발 단계의 가장 중요한 단계로 취급하고, 특히 사고에 의한 피해가 큰 안전성이 중요한 시스템의 경우에는 시스템을 운영하기 위하여 국내와 국제적인 규제 기관에서는

요구 명세의 분석을 통한 안전성의 입증을 시스템 공급업체에 요구한다[2].

SRS의 분석을 위해 제안된 방법들 중에 인스펙션[3,4]과 정형 방법이 효과적이라고 알려져 있다[5]. 인스펙션은 산업계에서 적용되어 좋은 결과를 보였고[3,4], 이상적으로 모든 종류의 오류를 찾아낼 수 있으며 월성 원자력 발전소 2,3,4호기의 제 2정지 시스템(SDS2: shutdown system number 2)을 인스펙션으로 검증한 결과를 평가했을 때에도 효과적이었지만[6], 인스펙션만으로 시스템을 검증한 후에는 시스템의 안전성 보장에 대한 확신에 한계가 있었다.

월성 SDS2의 SRS는 시스템의 외부에서 시스템으로의 입력을 의미하는 30개의 모니터 변수, 시스템에서 외부로의 출력을 의미하는 59개의 컨트롤 변수, 계산 기능을 표시하는 129개의 함수, 함수들 간의 계층 관계를 표시하기 위한 15개의 함수 그룹으로 구성되어 있다. SDS2의 SRS는 자료 흐름도와 유사한 그림과, 테이블 형식의 표로 기술되어 거의 400 쪽에 달한다. 그리고, 요구 명세 개발 기간의 마지막 해 1년 동안 4회의 작은 수정이 있었다. 이 SRS는 전산 처리를 위한 도구가 사용되지 않았기 때문에, 본 연구진은 기본적인 분석 방법으로 인스펙션을 선택할 수 밖에 없었다. 검증을 수행하는 시간은 인스펙션 전체 과정 중 인스펙션 미팅시간 만을 계산하여 80 man-hour가 걸렸으며, 17개의 사소한 오류가 발견되었다[6].

SRS를 인스펙션으로 검증하는 데에는 다음과 같은 어려움이 있었다. (1) 인스펙션 팀이 전산학 전공자로 원자력 관련 요구 사항을 완전하게 이해하기 어려웠다. 마찬가지로 원자력 전문가로 팀을 구성하는 경우에는 소프트웨어의 개념을 이해하기 어려웠을 것이다. 시스템 전체에 대해 완전히 이해 할 수 없었기 때문에 각 부분에 대하여 확인하는 정도로 머무는 경우도 있었다. (2) SRS가 이해되었다 하더라도 프로그램 기능 명세(PFS: Program Functional Specification)가 자연어로 기술되어있기 때문에 프로그램 기능 명세와 소프트웨어 요구 명세의 일관성의 분석을 체계화 또는 자동화하기 어려웠다. 또한, 요구 사항은 개발과 검증 기간에 계속적으로 변경되기 마련인데 그 변경의 영향이 변경되지 않은 부분에도 미칠 수 있으므로 명세에 대하여 다시 인스펙션을 수행해야 하는 어려움이 있었다. 반복적이고 유사한 작업을 반복하는 경우에 인스펙션을 수행하는 동안 인스펙션 팀은 쉽게 집중력이 떨어지고 자발적인 참여가 어려워졌다. (3) 보다 근본적인 한계는, 사람은 실수를 할 수 있는데 사람에 의해서 수행되는 인스펙션

의 특성상 인스펙션이 끝난 후에도 오류가 없다는 확신을 할 수 없었다. 따라서 인스펙션이 오류를 찾는 효과적인 방법이라는 것을 부인할 수 없지만 다른 검증 방법과 함께 사용되어야 한다는 결론을 내렸다.

이러한 경험을 통해 요구 명세의 검증 환경의 필요성을 인식하게 되었고, 요구 사항을 크게 시스템의 기능에 무관한 부분과 시스템의 기능에 관한 부분으로 나누어, 전자를 검사하는 환경을 구현하여 적용하였다[7]. 그리고, 본 논문에서는 후자의 검증을 위한 방법과 환경을 제안하였는데, 이 두 종류의 환경을 통하여 SCR (Software Cost Reduction)-style SRS의 전반적인 검증을 가능하도록 하였다. 기능에 관한 요구 사항을 검증하기 위한 방법으로 PFS와 SRS 간의 일관성 검증을 수행하는 환경을 개발하였다. 이 환경의 앞부분은 SRS의 편집과 분석을 위한 형태로 변환하는 부분이고, 뒷부분은 미국 SRI International에서 만든 정리 증명기인 PVS (Prototype Verification System)를 사용한 부분이다.

본 논문의 구성은 다음과 같다. 2절은 월성 SDS2의 요구 사항의 예를 통하여, 본 논문에서 대상으로하는 SRS의 구성과 PFS에 대해 설명하고, 검증 도구에서 사용하는 PVS에 대하여 설명하였다. 3절은 검증 환경과 변환 절차, 검증 방법을 설명한다. 또한 검증과 경험에 대한 기술을 하였다. 마지막으로 4절은 논문의 결론이다.

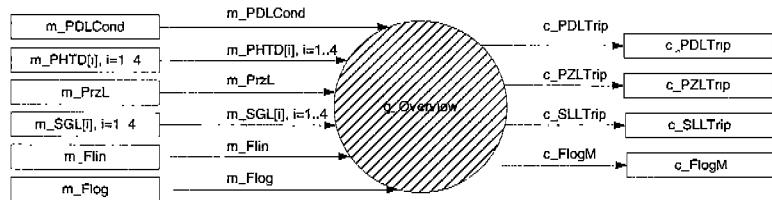
2. 관련연구

본 장은 시스템의 요구 사항을 기술한 방식인 SCR-style SRS, PFS와 정리 증명기 PVS에 대하여 기술한다.

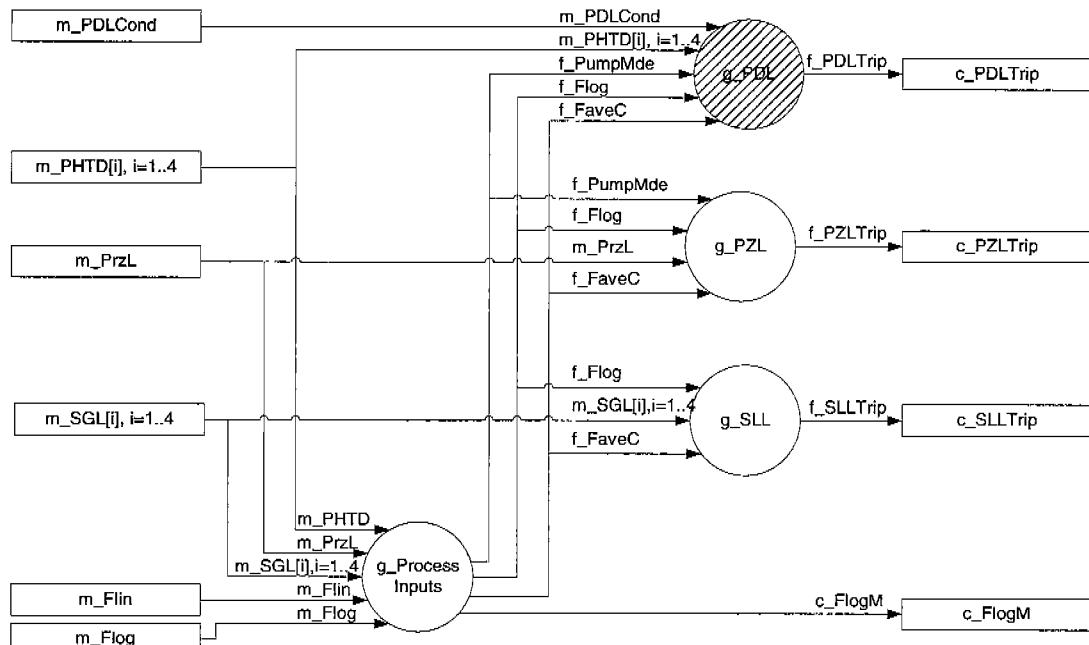
2.1 SCR-style SRS

본 연구의 검증 대상인 시스템은 SCR-style SRS이다. SCR-style SRS는 주기적으로 시스템의 외부 상태를 모니터하고 출력값을 계산하여 외부로 출력을 보내준다.

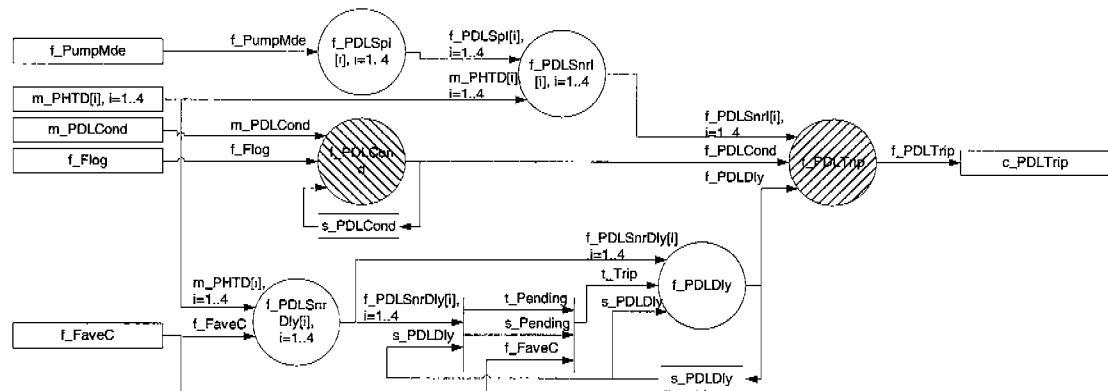
본 논문의 검증 대상인 월성 SDS2는 외부에서 원자로의 상태(예를 들어 압력, 온도, 운영 모드)를 센서를 통해 읽어들여 값(원자로 정지, 제어봉의 동작 위치)을 계산하여 작동기로 출력한다. SDS2는 PDC1 (Programmable Digital Comparator 1)과 PDC2의 두개의 유사한 소프트웨어 시스템으로 구성되어 있으며, PDC1은 30개의 센서로부터 값을 읽어 3개의 트립 신호를 생성한다. 3개의 트립 신호 중에 하나라도 트립 신호가 생성되면 원자로 시스템은 정지하며, 세 개의 트립 신호를 계산하는 시스템 중 가장 복잡한 것은 PDL 트립을 계산하는 시스템으로 본 논문에서의 검증 대상이다. SRS 전체의 분량은 374 페이지이고, 그 중 PDL 부분은 대략 20 페이지 정도의 분량에 해당한다.



(a) 월성 SDS2의 FOD의 일부



(b) g_Overview의 하위 수준 FOD



(c) g_PDL의 하위 수준 FOD

그림 1 Function Overview Diagram(FOD)의 일부분

```

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	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

그림 2 f_PDLCond의 매크로와 SDT

SCR-style SRS의 주요 구성 요소는 다음과 같다.

변수 선언: 시스템과 외부와의 인터페이스로서 모니터 변수와 컨트롤 변수로 기술된다. 모니터 변수는 컴퓨터 시스템으로의 입력을 의미하고, 컨트롤 변수는 컴퓨터 시스템으로부터의 출력을 의미한다. 각 변수의 정의를 위한 속성으로는 타입, 단위, 범위, 정밀도 등이 있다. 타입은 디지털 신호와 아날로그 신호로 구분된다.

Functional overview diagrams (FODs): FOD는 자료 흐름 도표와 유사한 표기로 자료의 흐름 뿐만 아니라 함수들의 계층관계를 표현할 수 있다. 즉, g_로 시작되는 그룹은 여러개의 그룹들 또는 함수 들로 세분화 된다. 가장 하위 수준에는 하나의 출력을 가지는 함수가 기술된다. 이 함수의 이름은 f_로 시작되며 출력되는 값도 동일한 이름의 변수의 값이다. 예를 들어 그림 1(a)는 소프트웨어 전체로의 입출력 관계를 기술하고 있는 g_Overview이다. g_Overview는 그림 1(b)의 g_ProcessInputs, g_PDL, g_PZL, g_SLL의 네개의 그룹으로 세분화된다. g_ProcessInputs는 입력에 대한 전처리 기능을 수행하고, g_PDL, g_PZL, g_SLL은 3종류의 트립 신호를 출력하는 시스템으로, 원전 시스템이 위험한 상태인지 설정된 값들과 비교하여 트립 신호를 생성한다. g_PDL 그룹은 다시 그림 1(c)의 9개의 함수와 2개의 타이밍 함수로 세분화되며, 함수 f_PDLTrip의 출력은 동일한 이름인 f_PDLTrip 변수의 값이다. 가장 하위 수준의 함수는 SDT (Structured Decision Table)와 타이밍 함수로 기술되며, 타이밍 함수는 FOD에서

바(|)로 표시된다. t_Pending과 t_Trip이 타이밍 함수이다. 특별한 종류의 함수로는 s_로 시작되는 상태 함수가 있다. s_PDLCond는 상태 함수의 예로, 이 함수는 f_로 표시된 함수의 값을 저장한 후 다음 번에 f_가 수행될 때 이용된다. 즉 값을 저장하는 기억 장소와 유사한 개념이다.

계층 구조뿐만 아니라 FOD는 그룹과 함수간의 자료 의존성이 표시된다. 이 자료 의존성에 의해 함수들의 계산을 수행하는 순서가 암시적으로 표시되어 있다. 예를 들어 그림 1(c)의 함수 f_PDLCond의 출력은 함수 f_PDLTrip의 입력으로 사용되므로, 앞의 함수가 계산된 후에만 뒤의 함수가 계산될 수 있다. 이것은 부분적 순서 관계에 의해 시스템의 계산 과정이 결정된다는 것이다. 이는 NRL의 SCR[8]과 LUSTRE[9]와 같은 자료 흐름 언어에서도 동일하게 나타난다.

Structured Decision Table (SDT): FOD의 하위 수준에 원으로 표시된 함수의 행동은 SDT라 불리는 테이블 형식으로 기술된다. 예를 들어 그림 2는 그림 1(c)에 짙은 색으로 표시된 함수 f_PDLCond를 정의한 것이다. 출력값은 일반적으로 상수 값이나 특정한 연산의 결과 값으로 정의된다. 그림 2의 경우 출력 값은 상수 k_CondOut 또는 k_CondIn으로 정의된다. k_로 시작되는 것은 상수를 의미한다.

SDT의 상부에 기술된 매크로는 특정 조건을 치환하는 역할을 한다. 예를 들어 그림 2의 2~5번째 줄은 w_FlogPDLCondLo[f_Flog]의 정의를 하고 있는데 f_Flog < k_FlogPDLLo - k_CondHys의 조건을 만족하면 3번째 줄의 표현에 의해 w_FlogPDLCondLo[f_Flog]의 값은 a이다.

SDT 부분은 AND-OR 테이블의 형태로 기술되는데 각 CONDITION STATEMENTS의 특정한 열의 각 행을 AND 한 것에 해당되는 ACTION STATEMENTS가 출력 값으로 지정된다. 그림 2의 경우 m_PDLCond = k_CondSwLo AND w_FlogPDLCondLo[f_Flog] = a인 경우 f_PDLCond는 k_CondOut을 출력으로 생성한다. 참고로 SDT에서 '-'는 무관 조건(don't care condition)을 의미한다.

타이밍 함수: 타이밍 함수는 시간 제약 조건을 기술하기 위해 필요하다. SCR-style SRS의 정의에 의하면 타이밍 함수 t_Wait은 t_Wait(C(t), Time_value, tol)로 되어있다. 시간 t에서의 논리적인 값을 의미하는 C(t)와 시간 간격 Time_value, 시간의 허용 오차를 의미하는 tol을 입력으로 받는다. 이 함수의 동작은 시간 t의 직전 시간 t'에서 출력 값이 false이고 t시간에서 입력 값

PHT Low Core Differential Pressure (PDL)	
1:	The PHT Low Core Differential Pressure (ΔP) trip parameter includes both an immediate and
2:	a delayed trip setpoint. Unlike other parameters, the ΔP parameter immediate trip low
3:	power conditioning level can be selected by the operator. A handswitch is connected to
4:	a D/I, and the operator can choose between two predetermined low power conditioning levels
5:	The PHT Low Core Differential Pressure trip requirements are:
6:
7:	e. Determine the immediate trip conditioning status from the conditioning level D/I as follows:
8:	1. If the D/I is open, select the 0.3% FP conditioning level.
9:	If $\psi_{10C} < 0.3\%FP - 50 \text{ mV}$, condition out the immediate trip.
10:	If $\psi_{10C} \geq 0.3\%FP$, enable the trip.
11:
12:	g. If no PHT ΔP delayed trip is pending or active then execute a delayed trip as follows:
13:	1. Continue normal operation without opening the parameter trip D/O for nominally
14:	three seconds.
15:	2. After the delay period has expired,
16:	open the parameter trip D/O
17:	if favec equals or exceeds 80%FP.
18:	Do not open the parameter trip D/O if favec is below 80%FP.
19:	3. Once the delayed parameter trip has occurred,
20:	keep the parameter trip D/O open for one second.
21:
22:	h. Immediate trips and delayed trips (pending and active) can occur simultaneously.
23:

그림 3 PFS(Program Functional Specification)의 일부

$C(t)$ 가 true인 경우, 출력 값이 true가 되며 출력 값 true를 시간 t 에서부터 Time_value동안 지속한다. Time_value 시간이 지난 직후에 출력 값을 false로 바꾸었다가, $C(t)$ 의 값을 검사하여 함수의 동작을 계속한다. 이상의 t_Wait의 정의를 이용하여 다양한 타이밍 함수를 정의할 수 있다.

그림 1(c)의 'l'로 표시된 함수인 t_Pending과 t_Trip가 타이밍 함수인데, t_Trip의 경우

$t_Trip = t_Wait(C, k_PDLTrip, k_PDLTripTol)$
where

$C = f_FaveC \geq k_Fave CPDL$
AND $t_Pending = \text{false}$
AND $s_Pending = \text{true}$

로 정의되어 있다. 즉 출력값이 false인 직후 조건 C인 $f_FaveC \geq k_Fave CPDL$ AND $t_Pending = \text{false}$ AND $s_Pending = \text{true}$ 가 만족되면 $k_PDLTrip$ 시간 만큼 true를 내보낸다. $k_PDLTripTol$ 은 시간의 허용 오차이다.

2.2 프로그램 기능 명세(PFS)

PFS(Program Functional Specification)는 시스템 전문가가 작성하며, 시스템 수준의 요구 명세라 할 수 있다. PFS는 특별한 형식없이 자연어로 쓰여 있으며 월성 SDS2는 영어로 쓰여있다. PFS는 소프트웨어 개발

단계의 가장 기본이 되는 문서이다. 그림 3은 프로그램 기능 명세의 일부이다. 월성 SDS2의 전체 PFS는 21페이지이고, PDL 트립에 관련된 PFS는 약 4페이지 정도의 분량이다.

2.3 PVS(Prototype Verification System)

PVS는 미국 SRI International에서 개발한 명세 작성과 증명 수행을 위한 대화형 도구[10]로, 이미 산업체에서 실제 적용되어서 우수성을 보였다. PVS의 성공적인 적용 사례로는 통신 프로토콜과 실시간 프로토콜같이 작은 규모의 시스템의 검증 뿐만 아니라[11,12,13,14], 거의 오십만 개의 트랜지스터로 구성된 항공용 마이크로 프로세서인 AAMP5의 검증[15], 항공기 제어 시스템의 검증[16] 등이 있다. 또한 TCAS II 명세에서 사용된 RSML 명세 언어의 상태들 간의 천이의 완전성과 일관성을 증명[17]하는데 사용되었다. PVS는 <http://pvs.csl.sri.com/>에서 무료로 소프트웨어를 받을 수 있으며, PVS의 사용을 결정한 근거는 다음과 같다.

- 본 연구진의 목표는 소프트웨어의 구조적인 측면, 기능적인 측면, 안전적인 측면의 결합을 위하여 실용적이면서 통합적인 소프트웨어 안전성 분석 환경을 개발하는 것이다. PVS는 현재 무료로 사용 가능한 정리 증명기로 이러한 통합 환경에서 중요한 부분이 될 것으로 생각하였다.

- PVS의 명세 언어는 고차 논리를 지원하면서 증명 과정에 매우 많은 자동화를 지원한다. 고차 논리를 통해 복잡한 시스템을 기술 할 수 있으며, 증명시에 기본적인 증명 명령어를 결합하여 강력한 증명 명령어를 정의하고 이를하여 많은 자동화를 지원한다. 예를 들어, 가장 많이 사용되는 증명 명령어 중 하나인 (grind) 명령은 PVS에서 모듈과 비슷한 개념인 THEORY의 정의를 읽어들여서 논리의 바꿔 적용(rewriting)과 반복적인 단순화를 통한 증명을 시도한다[14]. PVS는 (grind) 외에도 다양하고 강력한 증명 명령어를 제공한다.
- PVS는 SDT 표현을 지원하기 때문에 SCR-style SRS에서의 PVS로의 변환을 쉽게 해준다.

3. SCR-style SRS의 검증

3.1 검증 방법의 개요

본 논문에서 수행한 검증 방법의 개요는 그림 4에 나타나 있으며, 다음의 절차에 따른다.

(1) 본 연구에서 개발된 그래픽 에디터를 통하여 SCR-style의 SRS를 작성하고, 에디터의 변환기를 통하여 작성된 SRS에 해당하는 PVS 명세를 생성한다. 이 에디터는 Sparc 스테이션 상에서 GNU의 C++ 언어로 구현되었으며 Motif 환경을 통해 사용자 인터페이스를 제공한다. 이 도구의 실행 화면은 그림 5이다. 이 변환 단계는 자동화가 가능하며 3.2절에 기술하였다.

(2) 자연어로 기술된 PFS를 PVS 명세 언어로, 증명해야 하는 검사 조건으로 변환한다. 이 단계는 자연어로부터의 변환이기 때문에 완전한 자동화가 불가능하지만, 세밀한 참조표를 통해 변환에 도움을 주었고 유사한 검사

조건의 유사한 변환이 가능함에 차안하여 체계적인 작성이 가능하도록 하였다. 이 변환 단계는 3.3절에 기술하였다. 이 단계에서 작성된 PVS 명세와 이전 단계에서 작성된 PVS 명세가 결합되어 컴퓨터에 파일로 저장된다.

(3) PVS를 수행하고, PVS 명세 파일을 읽어서 각 검사 조건으로 커서를 옮긴 후 M-x pr 명령을 수행하면 증명을 위한 창을 열고 증명을 시작한다. 각 단계에서는 기존에 유사한 검사 조건을 증명할 때 사용했던 증명 명령어를 틀로 하여 증명을 수행할 수 있도록 하여 증명을 수행하는 사람의 부담을 줄였다. 이 검증 절차는 3.4절에 기술하였다. 이 후 검증한 결과를 분석하여 오류가 존재한다면 SRS나 PFS를 수정하여 다시 검증한다.

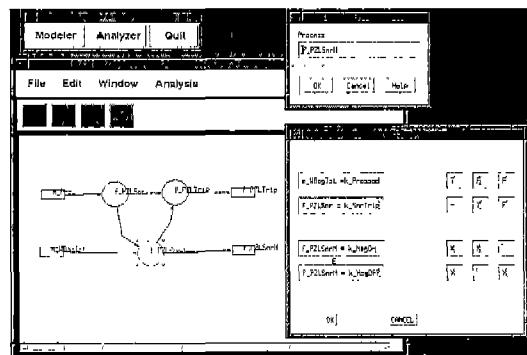


그림 5 SRS로부터 PVS 명세를 생성하는 도구

3.2 SCR-style SRS에서 PVS로의 변환

SCR-style의 SRS는 변수 선언, FOD, SDT, 타이밍 함수로 구성되어 있고, 검증을 위해서 각 부분은 PVS 명세언어로 변환되어야 한다. SCR-style SRS에서 4가지 구성 요소 외에 시스템의 동작의 이론적인 배경이 되는 시간 모델과 타입의 정의에 대한 변환이 추가적으로 필요하다. 본 절에서는 일반적인 변환 방법에 대한 설명과 함께 월성 SDS2의 SRS에 대하여 실제 변환 사례를 들어 설명하도록 하겠다. 논문에서 사용되는 변수와 상수는 대부분 실제 시스템에서 사용되는 것과 동일한 이름을 사용하였다.

변환은 1. 시간(tick) 모델 요소, 2. 타입의 정의, 3. 모니터 변수와 컨트롤 변수의 타입 선언, 4. 함수의 변환, 5. 타이밍 함수의 기본 정의와 타이밍 함수의 변환의 순서로 이루어지며, 이 부분을 모두 모은 내용이 SRS로 부터 변환된 PVS 명세의 일부분이다.

(1 단계) 시간 모델 요소의 선언:

SCR-style SRS에서는 시간은 자연수로 표시되며, 각

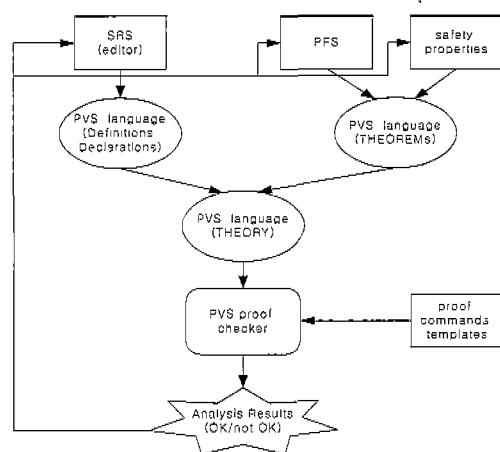


그림 4 SRS의 기능에 관련된 요구 사항의 검증 절차

연속적인 숫자는 주기의 증가하는 숫자를 의미한다. 이 부분은 SCR-style SRS라면 항상 동일하게 표현되며, 그림 6에 기술되어 있다. 첫 번째와 두번째 열의 '숫자:' 표시는 참조를 위한 행 번호로 실제 변환에서는 삽입되지 않는다. 시간의 흐름은 1행의 tick이란 타입으로 표시된다. tick은 nat으로 정의되었는데, 이는 PVS가 기본적으로 호출되는 라이브러리인 prelude.pvs에 정의 되어있으며 자연수를 의미한다. tick의 원소가 하나 이상이라는 것을 명확하게 표현하기 위해 TYPE+로 표시하였고, 그 포함된 원소가 0이라는 것을 표현하기 위하여 CONTAINING 0라는 구문을 첨가하였다. TYPE+와 CONTAINING은 본 논문에서 사용한 시스템의 경우에는 쓰지 않아도 오류가 발생하는 것은 아니지만 후에 타입 검사를 수행할 때 tick 타입의 변수가 실제 값이 존재하는 것을 증명할 필요가 있는 경우 선언시에 TYPE+와 CONTAINING으로 명시해주면 타입 검사를 자동으로 수행하기 때문에 증명 과정의 자동화가 가능해진다. 2행의 t는 tick 타입의 변수로, 변수를 선언할 때에는 VAR라는 키워드를 통해 선언된다. 3행은 초기값을 의미하는 상수인 init은 tick 타입의 0으로 선언된다. PVS에서 상수는 constant_ name : type = value의 형식으로 정의된다.

```

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

```

그림 6 (1단계) 시간 모델 요소의 선언

(2 단계) 타입, 상수 정의:

변수의 선언을 위해서는 타입이 정의되어야 한다. SCR-style SRS의 변수의 두개의 큰 범주는 아날로그 변수와 디지털 변수이다. 변수의 범위가 한정되지 않은 아날로그 변수의 경우 실수로 정의될 수 있고, 변수의 범위가 한정된 아날로그 변수의 경우에는 실수의 부분 타입으로 정의된다. 디지털 변수는 일반적으로 여러 개의 원소를 가지기 때문에 각 원소를 OR로 연결한 집합으로 정의한다. 변수의 값은 시간이 변화함에 따라 변화되기 때문에 tick에서 변수의 타입으로의 함수로 표시된다. 함수는 [와] 사이에 ->를 사용하여 대응 관계로 기술한다. 또한, 아날로그 타입이건 디지털 타입이건 한 개 이상의 값을 가지고 있는 것이 명확하다면 이것을 정의에 명확히 기술하는 것이 타입 검사의 자동화를 위하여 필요하다. 따라서 TYPE-와 CONTAINING을 사용하여 표시하여야 한다. 이때 tick으로 부터의 함수형 인 경우에는 lambda 키워드를 사용하여 매개 인자를

밝혀야한다. 추가적으로 상수에 대해서도 정의를 한다. 마지막으로 매크로에 의해 정의되고 SDT의 내부에서 참조되는 타입은 나열형으로 정의된다.

그림 7은 월성 SDS2의 SRS의 타입과 상수 정의의 일부이다. 1행은 가장 기본적인 타입인 milivolt의 정의로 아날로그 타입이었기 때문에 real로 정의된다. 2행은 milivolt의 정의를 사용하여 t_Milivolt를 tick으로부터 milivolt에 대응하는 함수 타입으로 정의한다. 4행은 디지털 타입을 표현하기 위하여 zero_one 타입으로 정의하였다. 집합으로 표현되었으며 적어도 0을 원소로 가지고 있음을 표현한다. 5행은 undef은 값이 정의되지 않은 상수로 정의하였다. 위래 SRS에서는 값이 정의되지 않은 상수가 존재하는 네, 이를 위해 undef를 사용한다. 6행과 7행은 k_Trip과 k_NotTrip의 상수를 정의하며 각각의 값은 0과 1이다. 8,9행은 k_CondIn과 k_CondOut을 정의하며 각각은 값은 정해지지 않은 상수이다. 11행에서는 k_Trip과 k_NotTrip의 상수를 사용하여 zero_one과 유사한 타입인 to_TripNotTrip을 정의한다. 12행과 13행에서는 tick에서 to_TripNotTrip으로의 함수인 t_TripNotTrip을 정의하였다. 이 타입은 적어도 t로부터 k_Trip으로의 상수가 포함된다는 것을 CONTAINING과 lambda 표현으로 정의하였다. 14행은 k_CondIn과 k_CondOut를 사용하여 to_CondInOut의 타입을 나열형으로 정의하였으며, 15행은 tick으로부터 to_CondInOut으로의 함수인 t_CondInOut을 정의하였다. 17행은 SDT의 내부에서 매크로로 사용될 enumabc의 정의이다. enumabc는 a, b, c의 원소를 가지는 나열형으로 정의하였다.

```

1: milivolt : TYPE = real           % 아날로그 변수
2: t_Milivolt : TYPE = [tick -> milivolt]
3:
4: zero_one : TYPE+ = {x:int | x=0 OR x=1} CONTAINING 0
   % 디지털 변수
5: undef : TYPE+
   % 값이 정해지지 않은 상수
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]
   % tick에서 to_TripNotTrip으로의
13:   CONTAINING lambda (t:tick) : k_Trip
   % 함수 타입, t->k_Trip 포함
14: to_CondInOut : TYPE = {k_CondIn, k_CondOut}
15: t_CondInOut : TYPE = [tick -> to_CondInOut]
16:
17: enumabc : TYPE = {a,b,c}

```

그림 7 (2 단계) 타입, 상수 정의

(3 단계) 모니터 변수와 컨트롤 변수의 타입 선언:

2 단계에서 선언된 타입을 이용하여 모니터 변수와 컨트롤 변수의 타입을 선언한다. 선언의 방법은 variable : type이다. 그림 8은 모니터 변수 m_Flog와 컨트롤 변수 c_PDLTrip의 선언의 예인데, 1행의 m_Flog은 t_Milivolt의 타입으로, 3행의 c_PDLTrip은 t_TripNotTrip의 타입으로 선언된다.

```

1: m_Flog : t_Milivolt      % 모니터 변수의 타입 선언
2:
3: c_PDLTrip : t_TripNotTrip % 컨트롤 변수의 타입 선언

```

그림 8 (3 단계) 모니터 변수와 컨트롤 변수의 타입 선언

(4 단계) 함수의 변환:

SCR-style SRS의 FOD는 함수 그룹을 계층적으로 세분화하면서 가장 하위 레벨에서는 타이밍 함수를 제외하고는 SDT로 구성된다. SRS의 검증을 위해 변환이 필요한 것은 가장 하위 수준의 함수를 표시한 SDT 만이 필요하므로, 가장 하위 레벨에서는 SDT를 통하여 내부의 함수를 표시하게 된다. 검증을 위해서는 실제 기능에 대한 부분만이 필요하므로 계층 구조에 대한 정보는 변환이 필요없고, SDT의 변환이 필요하다.

함수는 크게 두가지로 나누어서 생각할 수 있는데, 하나는 tick t에서의 다른 함수의 값을 받아서 계산하여 tick t에서 결과를 출력하는 함수이다. 즉, 계산 과정에서 시간이 걸리지 않는다는 가정을 하고 있다. 다른 하나는 다른 함수의 값뿐만 아니라 현재 계산하는 함수 자신의 이전 수행결과 즉 t-1에서 계산했던 값(상태 변수로 저장되어 있는 값)을 입력으로 받아 계산하여 결과를 출력하는 함수이다. f_X, f_A, f_B, s_X를 함수

또는 변수의 이름이라고 할 때, 첫번째 경우는

$$f_X(t) = \text{output}(f_A(t), f_B(t))$$

로 표시되어지고, 두번째 경우는

$$f_X(t) = \text{output}(f_A(t), s_X(t))$$

로 표시되는데 이때

$$t = 0 \text{ 인 경우 } s_X(t) = \text{initial_value},$$

$$t \neq 0 \text{ 인 경우 } s_X(t) = f_X(t-1)$$

로 표시된다. 두번째 경우는 함수의 정의 간에 원형의 존성이 존재한다. 즉 f_X를 정의하기 위해서는 output 함수의 입력으로 들어오는 s_X의 정의가 필요하고, s_X를 정의하기 위해서는 f_X의 정의가 필요하다. 이 원형

의존성은 tick t에서 s_X는 t-1에서의 f_X를 필요로하고, f_X는 tick t에서 t의 s_X의 값을 필요로 한다. 따라서, 서로 다른 tick에서의 값을 필요로 하기 때문에 값을 정의할 수 없는 원형 의존성이 아니기 때문에 의미적으로 문제가 없다. 하지만, PVS는 엄격하게 타입을 검사하기 때문에 이 경우에 타입에 대한 고려가 필요하다. PVS에서는 함수를 정의하는데 있어서 선언과 정의는 분리한 선언형(declaration style)과 함수의 재귀적 정의를 이용한 정의형(definition style)을 사용할 수 있다. 각각의 경우를 분리하여 설명하도록 하겠다.

선언형 변환:

선언형 변환은 변수의 계산 관계를 AXIOM 키워드를 사용하여 기술하는 방법이다. 이 방법은 3단계에서 모니터 변수와 컨트롤 변수의 타입 선언과 유사한 형태로 함수의 타입을 선언한 후 실제 함수의 입력과 출력 관계는 AXIOM이란 키워드를 사용하여 기술한다.

$f_X(t) = \text{output}(f_A(t), f_B(t))$ 이고, f_X 는 value_type 타입의 값을 출력으로 내보낸다고 가정하면,

- 1: axf_X : [tick -> value_type]
 - 2: axf_X : AXIOM $f_X(t) = \text{output}(f_A(t), f_B(t))$
- 으로 변환된다.

그리고 $f_X(t) = \text{output}(f_A(t), s_X(t))$, $t = 0$ 인 경우 $s_X(t) = \text{initial_value}$, $t \neq 0$ 인 경우 $s_X(t) = f_X(t-1)$ 에 대해서는

- 1: axf_X : [tick -> value_type]
- 2: axs_X : [tick -> value_type]
- 3: axf_X : AXIOM $f_X(t) = \text{output}(f_A(t), s_X(t))$
- 4: axs_X : AXIOM $s_X(t) = \text{IF } t = 0$
THEN initial_value
ELSE $f_X(t-1)$
ENDIF

로 변환된다.

선언형에서 함수의 타입을 선언한 것은 고급 프로그래밍 언어의 선언과 유사한 역할을 한다. 따라서, 3번째 행의 $f_X(t)$ 의 정의에서 아직까지 정의되지 않은 $s_X(t)$ 를 참조하고 있지만 타입 선언에 의해 문제가 없기 때문에 오류없이 통과한다. 또한 4번째 행의 $s_X(t)$ 에 대한 정의에서는 물론 1번째 행에서 $f_X(t)$ 에 대해 타입을 선언하고 있기 때문에 오류가 발생하지 않는다. 이상에서 본 것과 같이 선언형의 변환은 선언부와 함수 본체 부분으로 나뉘고, 특별한 분석과정 없이 PVS 명세 언어로 변환이 가능하기 때문에 변환이 용이하다. 하지만, 선언형으로 함수를 변환하는 경우에는 특정한 함수에 대하여 정의한 여러개의 AXIOM 들이 서로 모순이

될 수 있다. 예를 들어 위의

`axf_X : AXIOM f_X(t) = output(f_A(t), f_B(t))`

이 후에

`axf_X1 : AXIOM f_X(t) = output1(f_A(t), f_B(t))`

라는 정의가 추가로 삽입될 수 있는데 이 두개의 AXIOM들에 기술되어 있는 output과 output1이 상호 보순일 수 있다는 것이다. 국단적인 예로 output = NOT output인 경우가 있을 수도 있다. 이 경우 axf_X 와 axf_X1을 모두 만족하는 f_X(t)이 존재하지 않는다. 이러한 보순을 검사할 수 있는 기능을 PVS에서 제공하지 않기 때문에 일관성이 없는 AXIOM이 삽입될 수 있다. 또한, 증명 과정에서 f_X(t)를 치환할 필요가 있을 때, axf_X와 axf_X1 중 어느 것으로 치환을 할 지 결정하여야 하는데, 두 개의 정의가 동등한 위상을 가지고 있으므로 PVS가 자동으로 선택하지 못하고 사용자의 선택을 요구한다. 결과적으로 후에 시스템의 특성을 검증하는 과정에서 증명의 자동화에 한계가 있다.

월성 SDS2의 SDT중 그림 2 f_PDLCond의 선언형 변환은 그림 9이다. 그림 9의 선언형에서는 1행과 2행에서 볼 수 있듯이 f_PDLCond와 s_PDLCond의 타입을 정의하는 것으로부터 시작된다. 이전 단계에서 선언했던 타입을 이용하여 axf_PDLCond와 axs_PDLCond의 타입을 정의한다. 3,4행의 w_FlogPDLCondLo와 w_FlogPDLCondHi는 f_PDLCond의 정의의 내부에서 사용하는 변수를 선언하는 것이다. 정의의 내부에서는 타입과 변수의 선언이 불가능하기 때문에 내부에서 할 수 없기 때문에 정의의 외부인 이 위치에서 한 것이다. 5행부터 41행까지는 f_PDLCond(t)를 정의한 것이다. 6행에서 20 행까지의 LET .. IN 은 함수의 정의의 내부에서 사용하는 지역 정의로 매크로 정의를 위하여 사용된다. TABLE .. ENDTABLE은 테이블 표현을 지원하는 구문으로

TABLE

| cond1 | out1 ||

| cond2 | out2 ||

ENDTABLE

은 IF cond1 THEN out1 ELSE IF cond2 THEN out2 과 동일한 표현이다. %로 시작되는 문장은 주석문의 시작을 알리는 표시로 21-24, 26, 28, 30, 32, 34, 36, 38, 40 행은 테이블 형태를 읽기 쉽게 만들기 위한 주석이다.

정의형 변환:

정의형은 함수의 타입에 대한 선언없이 정의만으로 변환된 PVS가 구성되는 방식이다.

```

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

```

그림 9 f_PDLCond와 s_PDLCond의 선언형 변환

f_X(t) = output(f_A(t), f_B(t))이고, f_X는 value_type 타입의 값을 출력으로 내보낸다고 가정하면, 1: f_X(t:tick) : value_type = output(f_A(t), f_B(t))로 변환된다.

그리고, f_X(t) = output(f_A(t), s_X(t)), t=0 인 경우 s_X(t) = initial_value, t ≠ 0 인 경우 s_X(t) = f_X(t-1)에 대해서는

```

1: f_X(t:tick) : RECURSIVE value_type =
2:           LET
3:             s_X:[tick -> value_type] =

```

```

4:           LAMBDA (tt:tick)
5:             IF tt = 0 THEN initial_value
6:               ELSE f_X(tt-1)
7:             ENDIF
8:             IN
9:             output(f_A(t), s_X(t))
10:            MEASURE t
11: s_X(t:tick) : value_type = IF t = 0
12:               THEN initial_value
13:               ELSE f_X(t-1)
14:               ENDIF

```

로 변환된다.

함수의 정의의 경우 원형 의존 관계가 있는 경우에 원형 의존 관계에 있는 부분을 내부 선언으로 처리하여야 하는 어려움이 있다. f_X 의 정의인 1행에서 10행을 참조하면, f_X 의 내부인 9행에서 s_X 를 참조하는데, s_X 는 11행에서 14행까지 정의되어 있기 때문에, 9행에서 참조할 수 없다. 따라서 3행부터 7행까지 s_X 를 지역 정의로 하였다. 지역 정의에서는 인자를 가진 함수를 정의할 수 없고 하나의 템(term)에 대해서만 정의를 하고 있기 때문에, 11행의 $f_X(t:tick)$: $value_type$ 과는 다르게 $s_X[tick \rightarrow value_type]$ 으로 정의한다. 또한 f_X 내부인 6행에서 f_X 가 다시 참조되므로 재귀적인 정의이다. 따라서 1행에 f_X 가 재귀적인 정의임을 표현하기 위해서 RECURSIVE 키워드를 사용하여야 한다. 또한 PVS에서는 재귀적인 정의의 종료를 보장하는 것에 관한 검사 조건을 명시적으로 기술하도록 하고 있는데, 이는 내부에서 호출된 함수의 인자가 작아야 한다는 것으로 기술된다. 이를 위해 10행에서 MEASURE라는 키워드를 사용하여야 하며, 1행에서의 원래의 인자가 t 이고 6행의 f_X 의 인자가 $t-1$ 이기 때문에 MEASURE t 로 기술한다. 일반적으로 MEASURE에 기술되는 조건은 자동으로 생성할 수 없으나 SCR-style SRS에서는 항상 이전 주기 즉 tick t 에서 $t-1$ 일때의 값을 참조하는 형태로 나타나므로 MEASURE t 로 기술할 수 있다. 정의형으로 변환한 경우 PVS에서는 하나의 함수가 하나의 기능만을 정의하도록 한정되어 있기 때문에 동일한 이름의 함수가 다른 기능을 정의할 수는 없다. 따라서, 증명 과정에서 PVS가 함수의 정의를 확장할 때 유일하게 선택할 수 있다. 결과적으로 일관성없는 정의가 삽입될 가능성성이 없기 때문에 PVS의 다시 적음 기능이 충분하게 이용되어 증명 과정에 더 높은 자동화가 가능하다. 하지만, 원형 의존 관계에 있는 함수를 정의하기 위해서는 지역 정의와 재귀 함수의 정의를 사용하여야

하기 때문에 변환된 PVS 명세가 복잡해진다. 특히 여러 단계에 걸친 원형 의존 관계가 존재하는 경우 최악의 경우 한개의 함수는 원형 의존 관계에 나타나는 함수의 갯수만큼 추가로 지역 변수를 정의하여야 한다. 또한 정리 증명의 특성상 완전한 자동화가 불가능한데, 이 때 PVS에서 내부 정의는 정의를 펼친 형태로 증명 과정을 수행하는데, 이 부분이 실제 증명을 수행하는데 제어하기 어렵게 만들기도 한다.

원성 SDS2의 SDT중 그림 2 $f_{PDLCond}$ 의 정의형 변환은 그림 10이다. $f_{PDLCond}$ 에 대해서는 1행부터 40행까지 정의하였다. $s_{PDLCond}$ 에 대해서는 3행~6

```

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

```

그림 10 $f_{PDLCond}$ 와 $s_{PDLCond}$ 의 정의형 변환

행의 지역 정의와 43행~45행의 정의가 있다. 위에서 설명한 것과 같이 지역 정의에서는 `s_PDLCond : [tick -> to_CondInOut]`로 정의하였고, 외부에서는 `s_PDLCond(t:tick) : to_CondInOut`으로 정의하였다. 또한 재귀적인 정의가 필요하기 때문에 1행의 RECURSIVE와 41행의 MEASURE t로 기술되었고, 그 외의 부분은 정의형과 동일하다.

정의형 변환과 선언형 변환의 각 장점과 단점을 비교하면 다음과 같다. 정의형 변환은 증명 과정에서 선언형 변환 보다 더 많은 자동화를 제공하는 장점을 가진다. 하지만 변환된 명세가 복잡하고, 증명의 자동화 과정에서 자동화된 증명이 실패하는 경우에 증명을 진행하기 어렵다. 한편 선언형 변환의 경우에는 변환된 명세가 원래의 명세와 유사한 형태를 가지고, 증명 과정의 자동화를 적게 제공하지만 증명 과정을 증명하는 사람이 직접적으로 관리할 수 있는 장점이 있다. 따라서 우리는 시스템 명세의 초기 단계에서는 증명 과정을 잘 관리할 수 있고 변환된 명세가 직관적인 선언형으로의 변환을 추천하고, 시스템 명세가 대부분 작성된 단계에서는 증명 과정에서 더 많은 자동화를 지원하는 정의형 변환을 추천한다.

(5 단계) 타이밍 함수의 기본 정의와 타이밍 함수의 변환:

SCR-style SRS에서 나타나는 타이밍 함수를 위한 기본 정의는 그림 11이다. 1행~7행까지 정의되어 있는 `twf` 함수는 tick $t = 0$ 일때는 FALSE를 출력값으로 내고, $ts-1$ 에서 출력값이 FALSE이고 현재 검사 조건 ts 가 TRUE 일때 지정된 시간 간격 tv 동안 TRUE 값을 출력하는 것을 기술한 것이다. 이것은 타이밍 함수의 기본 기능이기 때문에 어떤 시스템의 명세에서도 들어가는 부분이다. 9행~10행에 기술된 `twfs`는 `twf`의 함수를 다른 함수와 마찬가지로 tick으로 부터의 출력 값(bool)으로의 함수 형태로 나타내기 위하여 기술한 것이다.

예를 들어, 타이밍 함수 `t_Trip`의 변환은 그림 12로 표시

```

1: twf(C:pred[tick], ttick, tv:tick): RECURSIVE bool =
2:   IF t = 0 THEN FALSE      % 초기값은 FALSE
3:   ELSE EXISTS (ts: (ttick | 0 < t)):
4:     (t-tv+1) <= ts AND ts <= t AND
          % 지정된 시간 간격에서
5:     (C(ts) AND NOT H(ts-1))
          % TRUE가 시작되는 시각이 있다면,
          % 출력값은 TRUE
6:   ENDIF
7:   MEASURE 1
8:
9: twfs(C:pred[tick], tv:tick) : [tick -> bool] =
10:  (LAMBDA (ttick):twf(H,C,t,ttick))

```

그림 11 (5 단계-1) 타이밍 함수의 기본 정의

된다. 그럼 12는 선언형으로 함수를 변환한 것이지만, 재귀적인 정의가 존재하지 않기 때문에 정의형으로의 변환도 쉽게 가능하다. 1행의 `axt_Trip`은 함수의 선언부이고, 4행~5행의 `axt_Trip`은 함수의 기능을 정의한 부분이다. 5행의 `cycletime`은 시스템이 동작하는 주기를 의미한다.

```

1: axt_Trip : [tick -> bool]
2: C_Trip(t:tick) : bool = f_FaveC(t) >= k_FaveCPDL AND
   (NOT t.Pending(t)) AND s_Pending(t)
4: axt_Trip : AXIOM t_Trip(t) =
5:           twfs(C_trip,k_trip/cycletime)(t)

```

그림 12 (5 단계-2) 타이밍 함수의 변환

3.3 PFS에서 PVS로의 변환

PFS가 PVS로 변환된 SRS와 일치하는지를 검사하기 위해서는 PFS로 PVS로 변환을 수행하여야 한다. 본 논문에서는 검사 조건을 PFS로부터 변환하였지만, FMEA(Failure Mode and Effect Analysis) 분석이나 관련 분야 전문가에 의해 작성된 안전성 검사 조건을 사용할 수도 있다.

PFS에서 PVS 명세 변환 과정은 크게 두 단계로 나눌 수 있는데, 첫 번째 단계는 PFS의 용어를 이미 PVS로 변환된 SRS의 용어와 일치시키는 단계이고, 두 번째 단계는 PFS의 문장을 의미상으로 동일한 PVS 명세로 바꾸는 단계이다. 비록 자동화가 될 수 없는 과정이고 PVS의 도움을 직접적으로 받은 것은 아니지만 본 논문에서는 체계적인 변환의 방법을 제안하였고, 첫번째 단계에서 PFS의 용어를 찾는 동안 일관적이지 않은 용어를 찾아줌으로써 PFS의 품질을 높일 수 있도록 하였다.

자연어로 기술된 PFS에 나타난 용어와 SRS에 나타난 용어들 간의 표현 방법이 차이가 있을 수 있다. 따라서, 용어와 실제 상수값등의 표현과 기호를 이미 변환된 SRS에 일치시키기 위하여 용어 간의 참조표를 구성하였다. 참조 표를 구성한 이후에 PFS를 SRS의 변수, 함수, 상수 등의 표현을 사용하여 PVS 명세언어로 표현하였다.

표 1 소프트웨어 요구 명세에 나타난 참조표

PFS	SRS
PHT Low Corc 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, t.PHTDAim[i], i=1..4, f_PHTDErr, f_PHTDM, f_PDLSpI[i], i=1..4, f_SprdChkA

처음에 PFS의 변환을 할때는 SRS 부록의 참조표를 이용하려고 하였다. 표 1은 참조표의 예이다. 이 표는 SRS와 PFS의 일관성을 추적하기 위해 작성된 것이다. 인스펙션으로 SRS를 검증할때에는 이 표가 도움이 되었다. 하지만, 이 표는 하나의 항목이 표현하고 있는 내용이 지나치게 많기 때문에 PFS로부터 PVS 명세로 변환시에 사용하기에는 적절하지 않다. 참조표의 하나의 항목은 프로그램 기능 명세의 3~4쪽의 분량이고 또한 소프트웨어 요구 명세는 대략 20~30쪽의 분량이다.

이 표를 대신하여 세밀한 참조표를 정의하였다. 이 세밀한 참조표는 프로그램 기능 명세의 단어 혹은 절과 소프트웨어 요구 명세의 변수간의 참조를 할 수 있도록 한 것이다. 이 참조표는 프로그램 기능 명세로부터 PVS의 THEOREM으로의 변환을 도와 줄 뿐만 아니라, 표의 작성 과정 중에 PFS의 일관성 없는 용어를 찾아준다.

표를 작성하는 과정은 SRS의 변수, 상수 등에 해당하는 PFS의 표현을 찾는 과정이다. 표를 작성하는 첫번째 단계에서는 SRS에 나와있는 각 변수의 설명 부분과 상수의 정의를 읽겨지는다. 즉 SRS의 설명을 찾아 그대로 읽겨지는다. 예를 들어 SRS에서 f_PDLTrip은 'Determine the state of PHT low core differential pressure parameter trip'로 설명되어있는데 이것을 읽겨 적는 것이다.

표를 작성하는 두번째 단계는 그림 3과 같은 PFS를 읽으면서 표의 항목에 해당하는 부분을 기술한다. 이 과정은 PFS의 의미를 이해하여야 하기 때문에 원자력 분야 전문가나 PFS를 이해할 수 있는 소프트웨어 공학자에 의해서 수행된다. 예를 들어 PFS의 ΔP_{trip} 과 parameter trip(D/O)이 f_PDLTrip과 일치한다는 것을 알아내어 표 2를 작성한다. 이 단계는 명세를 세밀하게 이해하면서 용어를 찾아내는 과정이기 때문에 이 과정에서 PFS 상의 일관적이지 않은 오류를 발견할 수 있다. 이 부분에 대해서는 3.5절에서 자세히 기술하겠다.

PFS는 자연어로 작성되어 있기 때문에 SRS의 각 부분에 해당하는 PFS의 표현을 찾았다고 하더라도 자동적인 변환은 불가능하다. 하지만 흔히 발견되는 문장의 종류를 구분하면 다음 3가지 경우가 있었다. 이외에도 추가적인 문장의 종류가 있을 수 있으나, 이 패턴 안에 있는 것이 대부분 이었다.

(패턴 1) 입력-출력 요구 사항으로 입력이 어떤 조건 일 때 출력으로는 어떤 값이 나온다는 것이다. 즉, 입력-출력 요구 사항은 항상 (즉, FORALL (t:tick)) $f_{condition}(t) = k_{condition}$ 인 경우 f_{output} 의 출력으

표 2 PFS로부터 PVS 명세로의 변환을 돋기위한 표

SRS	PFS
m_PDL Cond	Handswitch for selecting PHT low core differential pressure conditioning at 0.3 percent or 5 percent FP
	hand switch, low power conditioning level, conditioning level
m_PHTD	PHT core differential pressure signals numbered 1 to 4
	Core Differential Pressure measurement, ΔP_i , DP signal
f_Flog	Determine the log power in mv derived from ion chamber log power signal
	ψ_{LOG}
f_PDLCond	Determine the PHT low core differential pressure immediate trip conditioning status
	conditioning status, condition out the immediate trip/enable the trip
f_PDLTrip	Determine the state of PHT low core differential pressure parameter trip
	ΔP_{trip} , parameter trip (D/O)

로 k_{output} 을 내는 것을 기술한 것이다.

theorem_input_output : THEOREM

$$(f_{condition}(t) = k_{condition}) \Rightarrow f_{output}(t) = k_{output}$$

참고로, PVS에서는 한정되지 않은 변수의 경우, \forall 한정사를 첨가하여 처리하기 때문에 위의 THEOREM은

theorem_input_output:THEOREM FORALL(t:tick):

$$(f_{condition}(t) = k_{condition}) \Rightarrow f_{output}(t) = k_{output}$$

로 처리된다.

(패턴 2) 시간에 관련된 요구 사항으로 조건이 만족 된 후 특정 시간 간격 동안 출력을 유지하는 경우이다. tick t에서 $f_{condition}(t) = k_{condition}$ 이면 tick t에서 $t+time$ 까지 f_{output} 이 출력으로 k_{output} 을 낸다.

theorem_duration : THEOREM

FORALL (t:{ts:time|ts>0}) :

$$(f_{condition}(t) = k_{condition}) \Rightarrow$$

FORALL (ti: time):

$$((t <= ti \text{ and } ti <= t+time) \Rightarrow$$

$$f_{output}(ti) = k_{output})$$

(패턴 3) 시간에 관련된 요구 사항으로 조건이 만족 된 후 특정 시간 이후에 값이 발생하는 경우이다. tick t에서 $f_{condition}(t) = k_{condition}$ 이면 t에서 time이 지난 후에는 k_{output} 을 출력으로 낸다.

```

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

```

월성 SRS2 PFS에서 실제 변환 사례는 다음과 같다.
그림 3의 e.1 항목은, 'If the D/I is open, select the 0.3% FP conditioning level. If $\phi_{LOG} < 0.3\%FP - 50mV$, condition out the immediate trip. If $\phi_{LOG} \geq 0.3\%FP$, enable the trip.' 이다. 이 문장은 입력-출력 요구 사항으로 패턴 1에 해당한다. 'the D/I'는 그림 3의 3번째와 4번째 줄에서 'hand switch'에 연결되어 'low power conditioning level'을 선택할 수 있다고 되어있다. 따라서, 이 'the D/I'는 표 2에 의해 'm_PDLCond'에 해당하는 것을 알 수 있다. 이 후 표 2에 의해 'the D/I'가 open했다는 것을 m_PDLCond변수를 사용하여 기술하면 $m_PDLCond(t) = k_CondSwLo$ 일 때 임을 알 수 있다. 이 상태에서 $\phi_{LOG} < 0.3\%FP - 50mV$ 이면 'immediate trip'이 'condition out'된다. ϕ_{LOG} 는 f_Flog에 대응하고, $0.3\%FP - 50mV$ 는 SRS에서는 mv로 단위를 사용함으로 $0.3\%FP$ 를 mv로 나타내면 2739, 즉 k_FlogPDLLo로 변환된다. 이를 변수에 대한 정보는 PFS의 부록이나 SRS에서 찾을 수 있다. 이러한 조건 일때, immediate trip은 동작되지 않아야 한다 (condition out). 이를 변수를 통해 기술하면, $f_PDLCond = k_CondOut$ 이다. 이와같은 이해 과정을 통해 그림 13의 THEOREM th_e_1_1이 작성된다. 유사한 방법으로 $\phi_{LOG} \geq 0.3\%FP$ 일때 'enable trip'이라는 것은 $f_PDLCond(t) = k_CondIn$ 에 해당하여 THEOREM th_e_1_2로 작성된다.

```

th_e_1_1 : THEOREM
  % if the D/I is open, select the 0.3% FP conditioning level
  (m_PDLCond(t) = k_CondSwLo AND
  % if  $\phi_{LOG} < 0.3\%FP - 50mV$ ,
  f_Flog(t) < 2739-50)
  =>
  % condition out the immediate trip
  f_PDLCond(t) = k_CondOut
th_e_1_2 : THEOREM
  % if the D/I is open, select the 0.3% FP conditioning level
  (m_PDLCond(t) = k_CondSwLo AND
  % if  $\phi_{LOG} \geq 0.3\%FP$ ,
  f_Flog(t) >= 2739)
  =>
  % enable the trip
  f_PDLCond(t) = k_CondIn

```

그림 13 프로그램 기능 명세로부터 PVS THEOREM으로의 변환 예

3.4 검증

이상과 같이 SRS는 선언형 또는 정의형의 PVS 명세로 변환되고, PFS는 증명할 특성을 기술한 THEOREM으로 변환된다. 이 변환된 각 부분은 THEORY라고 하는 PVS의 파일을 구성한다. 이후, THEORY의 THEOREM은 PVS를 사용하여 검증한다. 검증 결과 PFS나 SRS의 오류가 존재한다면 수정한다. 물론 경우에 따라서 PFS를 THEOREM으로 변환하는 과정에서 잘못한 경우도 있다. 이 때는 THEOREM을 수정한다. PVS에서 증명 과정은 증명을 원하는 THEOREM 위에 커서를 옮겨 놓고, M-x 키를 누르면 PVS의 명령어를 입력할 수 있는 명령 모드가 시작되는데, 여기에 pr 을 치면 증명을 위한 새로운 창이 열리면서 증명을 시작하게 되고, PVS의 증명 명령어를 입력하면서 증명이 진행되게 된다.

PVS는 정리 증명법을 지원하는 정리 증명기이다. 정리 증명법은 증명 과정이 자동화 될 수 없고, 사용자의 개입이 필요하기 때문에 어렵다는 비판을 받아왔다. 하지만, 본 연구에서 실제 검증을 수행한 결과 THEOREM으로 표시된 유사한 패턴의 검사 조건의 경우 유사한 패턴의 검증 명령어가 사용된다는 것을 발견하였다. 즉, 검증의 자동화를 가능하게 하는 패턴이 있음을 발견하였다. 즉, 구체적으로 어떤 THEOREM을 증명할 때, 이와 유사한 형태의 THEOREM을 증명할 때 사용했던 증명 명령어와 유사한 명령어를 사용할 수 있다.

선언형의 경우 자동적인 다시 적음이 불가능하기 때문에 함수의 정의를 (lemma "...")를 통해 증명 과정의 가정 부분으로 읽어들이고, 필요한 경우 (inst?) 명령으로 변수를 치환 (instantiation) 하고 (replace "...")로 등식의 다시 적음을 수행하고 (grind) 명령으로 증명을 완료할 수 있다. 물론 경우에 따라서 replace를 통하여 한번 이상의 다시 적음을 수행해야하는 경우도 있다. "..." 부분은 FOD의 데이터 흐름 경로 상에 존재하는 함수나 정의에 관련된 것이다. THEOREM에 영향을 미칠 수 있는 부분들 만이 실제 증명에서 필요하다는데에 기반하여 데이터 흐름 경로 상에 있는 변수나 함수를 기준으로 하여 다시 적음을 수행하는 것이다. 하지만, 실제 증명을 수행하면서 데이터 흐름 경로 상의 정확히 함수를 선택할 것인가는 어려운 문제이다. 그림 13의 THEOREM th_e_1_1과 THEOREM th_e_1_2의 경우 (lemma "axf_PDLCond") (grind)로 증명을 마칠 수 있다. 증명을 성공적으로 끝낸 경우에는 PVS는 QED란 메시지를 내보내고 증명을 끝낸다. 일반적으로 선언형인

경우에는 자동적인 다시 죽음이 불가능하기 때문에 증명 과정 중에 사람의 개입이 많이 요구된다. 하지만, 증명 과정을 사람이 전적으로 제어할 수 있기 때문에 명세에 대한 이해는 그만큼 더 높아진다.

정의형으로 SRS가 PVS 명세로 변환된 경우는 선언형에 비하여 증명에 보다 많은 자동화가 가능하다. 정의형의 경우 증명 명령어는 크게 (expand "...") (grind :exclude "..."))와 (grind :exclude "..."))의 두가지 범주로 나뉜다. 명령어의 각 부분을 설명하면 다음과 같다. expand는 정의된 함수를 확장하는 역할을 한다. 경우에 따라서 재귀적인 정의를 한 번은 풀어쓸 필요가 있는데 이를 위해서 expand 명령을 사용한다. "..." 부분은 FOD의 데이터 흐름 경로 상에 존재하는 함수나 정의에 관련된 것이다. 이는 THEOREM에 영향을 미칠 수 있는 부분들 만이 실제 증명에서 필요하다는데에 기반하여 데이터 흐름 경로 상에 있는 값을 만이 필요하다. grind는 가장 강력한 증명 명령어 중 하나로 바꿔 죽음, 단순화, 치환 등을 가능하면 많이 진행한다. 하지만, grind는 다시 죽음이 가능하다면 제한 없이 계속 수행하기 때문에, 함수의 정의가 재귀적인 경우에 무한한 다시 죽음을 수행한다. 따라서, exclude의 ... 부분에 재귀적으로 정의된 함수를 적어줌으로써 그 함수의 다시 죽음을 제한으로써 무한한 다시 죽음을 막아준다. 정리하면 grind 명령을 exclude와 expand를 통하여 적절히 조절함으로써 증명을 가능하도록 한다. 본 논문에서 이러한 증명 방법이 정형적으로 제안되지는 않았지만, 실제 증명과정에서 적용하면 상당한 효과를 볼 수 있다. 그럼 13의 THEOREM th_e_1_1과 THEOREM th_e_1_2를 증명한다면 이 THEOREM에 나타나는 변수들의 자료 흐름을 참조하여, f_PDLCond가 재귀적인 정의임을 알 수 있다. 이 후 (expand "f_PDLCond") (grind :exclude ("f_PDLCond"))를 증명 명령어로 사용하여 f_PDLCond를 expand로 확장하고, exclude로 f_PDLCond의 확장없이 grind를 통해 증명을 완료할 수 있다. 정의형의 경우 증명의 자동화가 가능한 부분이 많이 있으나, 실제 검증한 결과로는 증명 과정을 제어하기에 어려움이 있었다. 특히, 정리 증명의 특성상 완전한 자동화가 불가능한데, 이때 PVS에서 내부 정의는 정의를 펼친 형태로 증명 과정을 수행하는데, 이 부분때문에 실제 증명을 수행하는데 제어하기 어려운 경우가 있었다.

3.5 검증 결과

PVS로 월성 SDS2의 요구 사항을 검증하는 과정에서 일관적이지 못한 용어의 사용, 갑작진 가정, 기능 요

구 사항으로 기술이 어려운 프로그램 기능 명세의 항목들을 발견하였다.

첫째, PFS에 일관적이지 못한 용어의 사용이 확인되었다. 변환을 위해서 참조표를 구성하는 과정에서 발견된 오류가 있었는데, 이것은 PFS의 용어가 일관성 없이 사용되었다는 것이다. 이 오류는 PVS로 검증을 수행하여 발견하는 것이 아니라 변환 과정에서 찾아지는 오류이다. 예를 들어 표 2의 1번째 항목인 m_PDLCond는 hand switch, low power conditioning level, conditioning level 등으로 사용되었고, 2번째 항목인 m_PIHTD는 Core Differential Pressure measurement, ΔP_i , DP signal 등으로 사용되었다. 또한, 5번째 항목인 f_PDLTrip은 PFS에서 the state of PHT low core differential pressure parameter trip, ΔP_{trip} , parameter trip(D/O) 등으로 사용됨을 알 수 있다. 이와 같이 일관적이지 않은 표현은 월성 SDS2의 PFS에서는 심각한 오류라고는 할 수 없지만, 자칫 PFS를 잘못 이해하거나 혼동할 수 있는 요인이 될 수 있다. 따라서, 시스템을 기술하는 사람은 일관적이지 않은 용어를 하나의 용어로 교체하여야 한다. PFS에서 다른 모호한 용어 사용의 예는 'condition out the immediate trip'과 'enable trip'이다. 자칫 'enable trip'을 'condition out'의 반대 의미가 아닌 trip 신호를 내보내는 것으로 이해할 수 있다. 따라서, 이런 부분들도 명확하게 수정하여야 한다. 그림 14는 수정된 PFS의 부분이고, 고쳐진 부분은 이탤릭체로 표시하였다. 그림 3의 e. 의 'the conditioning level'을 'the low power conditioning level'로 구체적으로 고쳤고, 'condition in - enable'의 쌍 대신 'disable - enable'이란 표현으로 수정하였다. 물론, 'condition in - condition out'이란 표현으로 수정하는 것도 무관하다.

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 $\phi_{LOI} < 0.3\%FP-50 mV$, disable the immediate trip.
 If $\phi_{LOI} \geq 0.3\%FP$, enable the immediate trip.

그림 14 모호성을 제거한 PFS의 부분

둘째, PFS에 숨겨진 가정이 있는 경우가 있다. 예를 들어 그림 3의 g.2와 g.3 항목을 그림 15의 PVS 명세로 변환하였다. 참고로 g.3 항목은 3.3절에서 분류한 패턴 중 (2) THEOREM theorem_duration을 기본으로 사용하여 기술하게 된다. 그림 15에서 cycletime (milliseconds)이란 SRS에서 가정하고 있는 시스템의

동작 주기를 반영하여 tick t를 실제 시간으로 변환하면 $t \times \text{cycletime}$ (milliseconds)이다. 따라서 1 seconds 를 표시하기 위하여 1000/cycletime로 기술한 것이다. 이후, 그림15의 THEOREM th_inappropriate_g_3의 증명에 실패하였다. 실패한 원인을 조사해 보니 이 요구 사항에는 숨겨진 가정이 있었다. 그림 3의 항목 g.2와 g.3은 대등한 관계이고 서로 독립적인 기능을 기술한 것처럼 보이지만, 실제로는 서로 의존적이다. 즉, 항목 g.2.1과 g.2.2가 성립 될때만 항목 g.3이 성립된다. 다시 살펴보면 'Once the delayed parameter trip has occurred'가 단순하게 'the delayed parameter trip has occurred'를 의미하는 것이 아니라 3 초간의 delay 후 'f_AVEC equals or exceeds 80%FP' 인 경우에 'the delayed parameter trip has occurred'이 만족되고 이 때 트립 신호가 1초간 지속된다는 것이다. 따라서, 항목 g.3의 조건 underline{the delayed parameter trip has occurred}는 g.2.1과 g.2.2에 기술된 내용을 암시적으로 가정하고 있다. 결과적으로, 위의 요구 사항은 그림 16에 수정된 것과 같이 THEOREM th_appropriate_g_3

```

th_appropriate_g_2_1 : THEOREM FORALL (t:{ts:time|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:time|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:time|ts>0}):
  t_Trip(t-1) = false AND t_Trip(t) = true AND
  => FORALL(t1 : time):
    ( (t <= t1 and t1 <= 1000/cycletime +t) =>
      t_Trip(t1) = true)

```

그림 15 PFS의 잘못된 변환 예

```

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 and _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)

```

그림 16 그림 15의 잘못된 변환의 수정 예

에 기술되어야 한다. 즉, 위에서 언급한대로 $f_FaveC(t) \geq 80$ AND $t_Pending(t) = \text{false}$ AND $s_Pending(t) = \text{true}$ 가 조건으로 추가되어야 한다. 사실상 이 오류는 인스페션 과정에서도 발견되지 않은 오류로 정형적인 검증을 수행하지 않으면 발견하기 어렵다.

4. 결 론

구조에 관련된 요구 사항과 기능에 관련된 요구 사항의 자동화된 검증을 위한 그래픽으로 사용자 환경이 제공되어지는 소프트웨어 도구를 개발하였다. 이 도구는 SRS를 PVS 명세로 변환하여 주는 기능도 함께 제공한다. 또한, PFS를 PVS의 THEOREM 틀로 변환해 줌으로써 기능에 관련된 요구 사항도 검증을 할 수 있다. 이 과정은 완전히 자동화가 되지 않고 사람에 의하여 변환되어야 하지만, 참조표를 사용하여 체계적인 변환이 가능하도록 하였다.

본 논문에서 제시한 방법의 사용자는 일반적인 정리 증명법으로 증명을 수행하는 것 보다는 PVS에서 제공하는 증명의 자동화와 증명 패턴을 사용하여 정형 방법론과 PVS의 전문가가 아닌 사용자에게 도움을 줄 수 있도록 하였다. 개발된 편집기는 소프트웨어 요구 명세를 편집하면 자동으로 PVS 명세언어로 변환을 하기 때문이다. 증명과정에 있어서는 증명 패턴을 이용해서 PVS의 약간의 지식만으로 증명을 쉽게 할 수 있다. 물론 복잡한 의존 관계가 있는 경우 정리 증명 방법과 PVS에 대한 증명하기가 어렵다. 더구나 시스템 내부에 유한 상태 기계의 특성이 있는 경우 PVS만을 사용해서 증명을 수행하는 것은 증명의 자동화의 한계가 있다. 따라서, 차후에는 자동화된 검증 방법으로 많은 연구가 되었던 모델 검사법과의 통합된 환경하에서의 검증이 필요하다고 생각된다.

마지막으로 다음과 같은 향후 연구가 있을 수 있으며, 현재 개발 중인 다른 종류의 워크 시스템의 제어 시스템에 적용 계획이다.

- 보다 많은 구조에 관련된 요구 사항의 정의와 PVS의 변환, 그리고 검증 명령어
- 본 논문에서 대상으로 한 SRS 외에 statecharts나 SCR같은 명세 언어로부터의 변환
 - 자연어로 쓰여진 프로그램 기능 명세로 부터 PVS THEOREM으로의 참조표 이상의 보다 체계적인 변환
 - 증명 명령어의 패턴에 대한 연구와 모델 검사법 같은 검증 방법과의 통합을 통한 증명의 자동화
 - 사용하기 보다 편리한 편집기의 재 구현

참 고 문 헌

- [1] R. Lutz, "Targeting Safety-Related Errors during Software Requirements Analysis," *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 99-106, 1993.
- [2] D. S. Herrmann, *Software Safety and Reliability*, IEEE Computer Society, pp. 1-503, 1999.
- [3] M. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, 12(7), pp. 133-144, 1986.
- [4] D. Wheeler, B. Brykczynski, and R. Meeson, Jr., *Software Inspection: An industry best practice*, IEEE Computer Society Press, pp. 1-312, 1996.
- [5] M. Hinckey and J. Bowen, *Application of Formal Methods*, Prentice-Hall, pp. 1-442, 1995.
- [6] T. Kim, H. Hong, S. Cho, W. Chun, and S. Cha, "A Verification of Requirements Specification for Safety-critical System," *22th KISS Spring Conference*, April, 1996.
- [7] T. Kim, and S. Cha, "Automated Structural Analysis of SCR-style Software Requirements Specification using PVS," *Journal of Software, Testing, Verification, and Reliability*, 11(3), pp. 143-163, 2001.
- [8] 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.
- [9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, 79(9), September 1991.
- [10] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivastava, "A Tutorial Introduction to PVS," *Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95)*, pp. 1-112, 1995.
- [11] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, *PVS System Guide Version 2.3*, Computer Science Laboratory, SRI International, pp. 1-88, 1999.
- [12] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert, *PVS Language Reference Version 2.3*, Computer Science Laboratory, SRI International, pp. 1-87, 1999.
- [13] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architecture: Prolegomena to the Design of PVS," *IEEE Transactions on Software Engineering*, Vol. 21, No. 2, pp. 107-125, 1995.
- [14] N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert, *PVS Prover Guide Version 2.3*, Computer Science Laboratory, SRI International, pp. 1-117, 1999.
- [15] S. Miller and M. Srivastava, "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)*, pp. 2-16, 1995.
- [16] B. Dutertre, and V. Stavridou, "Formal Requirements Analysis of an Avionics Control System," *IEEE Transactions on Software Engineering*, 23(5), pp. 267-278, 1997.
- [17] M. Heimdahl and B. Czerny, "Using PVS to Analyze Hierarchical State-Based Requirements for Completeness and Consistency," *Proceedings of the IEEE High Assurance Systems Engineering Workshop (HASE '96)*, pp. 252-262, 1996.

김 태호



1995년 성균관대학교 정보공학과 학사.
1997년 한국과학기술원 전산학과 석사.
1997년 ~ 현재 한국과학기술원 전자전
산학과 전산학전공 박사과정. 2001년 ~
현재 SRI International, Menlo Park,
CA, USA 방문연구원

차 성덕



1983년 University of California, Irvine 학사. 1986년 University of California, Irvine 석사. 1991년 University of California, Irvine 정보 및 전산학 박
사. 1990년 ~ 1991년 Hughes Aircraft Company, Ground Systems Group,
Fullerton, CA, USA. 1991년 ~ 1994년 The Aerospace
Corporation, El Segundo, CA, USA. 1994년 ~ 현재 한
국과학기술원 전자전산학과 전산학전공 교수